ValinaGAN_DCGAN

December 21, 2020

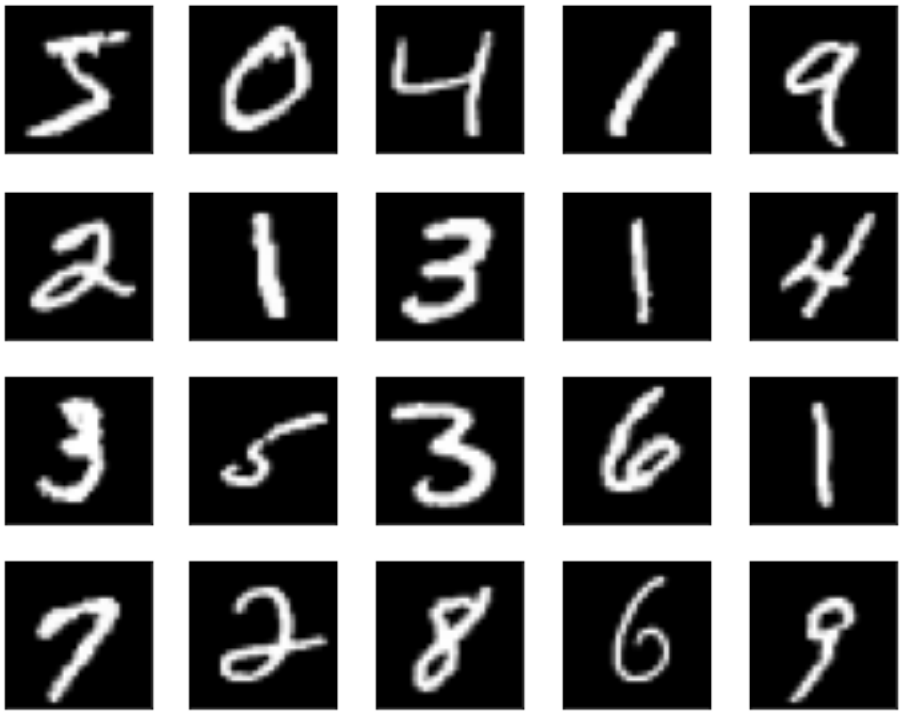
1 1. Valina GAN (MNIST)

[ ]:
```
import numpy as np
import keras
from keras.layers import Input, Dense, Activation, LeakyReLU, BatchNormalization
from keras.models import Sequential
from keras.optimizers import Adam
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
%matplotlib inline
```

First, we download MNIST dataset and examine some sample images.

[ ]:
```
(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()

plt.figure(figsize=(5, 4))
for i in range(20):
  plt.subplot(4, 5, i+1)
  plt.imshow(X_train[i], cmap='gray')
  plt.xticks([])
  plt.yticks([])
plt.tight_layout()
plt.show()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
```



1.1 1.1 Make Valina-GAN

The input to the generator is called 'latent sample' which is a series of randomly generated numbers. We use the normal distribution here.

Hyperparameters: sample size;

[ ]:
```
def make_latent_samples(n_samples, sample_size):
  return np.random.normal(loc=0, scale=1, size=(n_samples, sample_size))
```

we use a vector of 100 randomly generated number as a sample

[ ]:
```
# generates one sample
make_latent_samples(1, 100)
```

[ ]:
```
array([[-0.73768705,  0.79707071, -0.19723314, -0.54128168, -0.55715322,
         2.26974179, -0.98732237, -1.66048143,  0.47233296,  0.16470538,
        -0.81117349, -0.32982775,  1.2131049 ,  0.66448386,  0.1379414 ,
         0.84696316, -1.53583959, -1.26424969, -0.27715885, -0.0100119 ,
        -0.10306932, -2.07962833,  0.21385855,  0.11793262, -1.97201689,
        -1.71641239,  1.31803346, -0.64117829, -1.14228291,  1.09332922,
        -0.47621838,  0.61610291, -1.732484  ,  0.95488777,  0.7932905 ,
         0.09381961,  0.55643936, -0.89053078, -0.52851243,  0.55665139,
        -0.18676647, -0.02677839, -1.31262348, -0.52402141,  0.13001672,
```

```
-0.85753881, -0.07136637,  1.053895  , -0.0607594 , -3.02710123,
-1.36985447, -0.90021885, -0.67593736, -0.85416076,  0.46737184,
 0.56263178,  0.31215079, -1.33387012,  0.12306829,  0.41466196,
-0.47334018, -0.30078187,  0.59733319, -0.04593709, -1.7020249 ,
 0.5670611 , -0.01028516, -0.91380671,  0.34401182,  1.35893263,
-1.1059757 ,  1.89813961, -0.95657472,  1.04386646,  1.76553412,
 2.48808497, -1.1278457 ,  0.23755172,  0.47041317, -0.3853109 ,
 1.624289  ,  1.18412902,  0.50530992, -0.1207714 , -1.17003678,
 0.18679796,  0.82450462, -0.18293209, -0.34769521,  0.48982245,
 0.25834508,  1.08764759, -0.32099137,  1.04597952,  0.65946054,
 0.04713836,  0.13896694, -0.56040365,  1.89390169, -0.61576633]])
```

- The below function combines the generator, discriminator, and GAN models and also compile them for training.
- The architecture of **Generator**:

1. Dense layer: *g_hidden_size* units, take one latent sample (100 values) as input;
2. LeakyReLU activation;
3. Dense layer: produces 28*28=784 data points which represent a digit image; we use 'tanh' as activation here since it is one of the best choices according to [1].

- The architecture of **Discriminator**:

1. Dense layer: *d_hidden_size* units, take 784 data points as input;
2. LeakyReLU activation;
3. Dense layer: The last activation is sigmoid to tell us the probability of whether the input image is real or not.

- **GAN**:

We connect the generator and the discriminator to produce a GAN. It takes the latent sample, and the generator inside GAN produces a digit image which the discriminator inside GAN classifies as real or fake.

[1] ***How to Train a GAN? Tips and tricks to make GANs work.*** Facebook AI Research: Soumith Chintala, Emily Denton, Martin Arjovsky, Michael Mathieu

[ ]:
```
def make_valina_GAN(sample_size, g_hidden_size, d_hidden_size, leaky_alpha,␣
↪g_learning_rate, d_learning_rate):
 keras.backend.clear_session()

 generator = Sequential([
     Dense(g_hidden_size, input_shape=(sample_size,)),
     LeakyReLU(alpha=leaky_alpha),
     Dense(784,activation='tanh')
 ], name='generator')

 discriminator = Sequential([
     Dense(d_hidden_size, input_shape=(784,)),
     LeakyReLU(alpha=leaky_alpha),
```

```
    Dense(1, activation='sigmoid')
], name='discriminator')

gan = Sequential([
    generator,
    discriminator
])

discriminator.compile(optimizer=Adam(lr=d_learning_rate),␣
↪loss='binary_crossentropy')
gan.compile(optimizer=Adam(lr=g_learning_rate), loss='binary_crossentropy')

return gan, generator, discriminator
```

1.2 1.2 Train Valina GAN

Preprocess:

We need to flatten the digit image data as the fully connected input layer expects that. Also, as the generator uses the tanh activation in the output layer, we scale all the MNIST images to have values between -1 and 1.

[ ]:
```
def preprocess(x):
  x = x.reshape(-1, 784) # 784=28*28
  x = np.float64(x)
  x = (x / 255 - 0.5) * 2
  x = np.clip(x, -1, 1)
  return x
```

[ ]:
```
X_train_real = preprocess(X_train)
X_test_real  = preprocess(X_test)
```

Deprocessing:

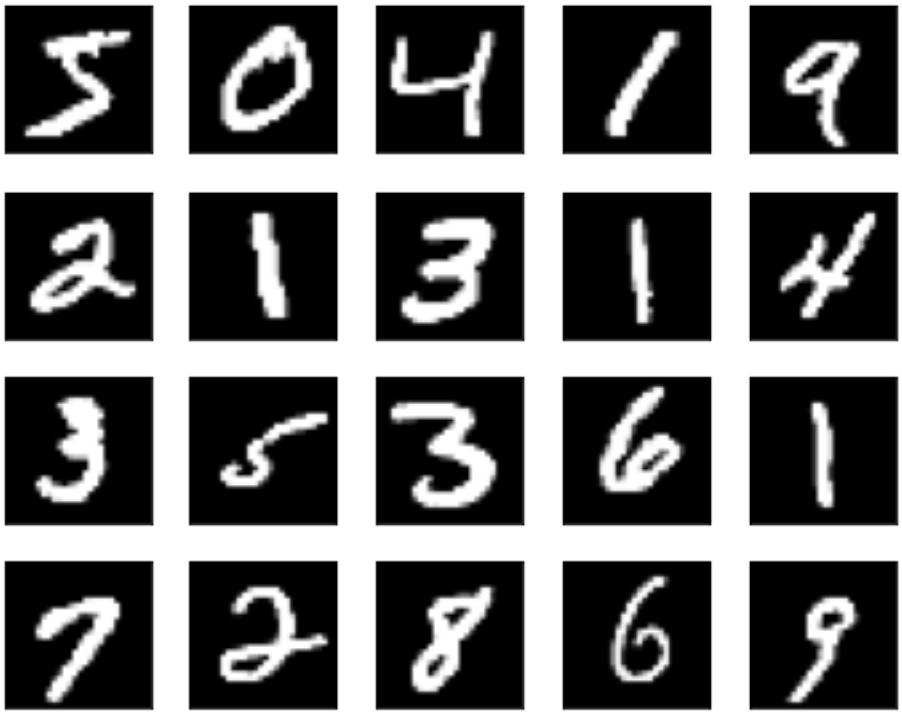
We also need a function to reverse the preprocessing so that we can display generated images.

[ ]:
```
def deprocess(x):
  x = (x / 2 + 1) * 255
  x = np.clip(x, 0, 255)
  x = np.uint8(x)
  x = x.reshape(28, 28)
  return x
```

Let's examine these two functions on some images:

[ ]:
```
plt.figure(figsize=(5, 4))
for i in range(20):
  img = deprocess(X_train_real[i])
  plt.subplot(4, 5, i+1)
```

```
  plt.imshow(img, cmap='gray')
  plt.xticks([])
  plt.yticks([])
plt.tight_layout()
plt.show()
```



[ ]:
```
#The labels are 1 (real) or 0 (fake) in 2D shape.
def make_labels(size):
  return np.ones([size, 1]), np.zeros([size, 1])
```

While training the GAN, the back-propagation should update the weights of the generator but not the discriminator.

As such, we need a way to make the discriminator trainable and non-trainable.

[ ]:
```
# make trainable
def make_trainable(model, trainable):
  for layer in model.layers:
    layer.trainable = trainable
```

Training Loop

We repeat the following to make both the discriminator and the generator better and better:

- Prepare a batch of real images

- Prepare a batch of fake images generated by the generator using latent samples
- Make the discriminator trainable
- Train the discriminator to classify the real and fake images
- Make the discriminator non-trainable
- Train the generator via the GAN

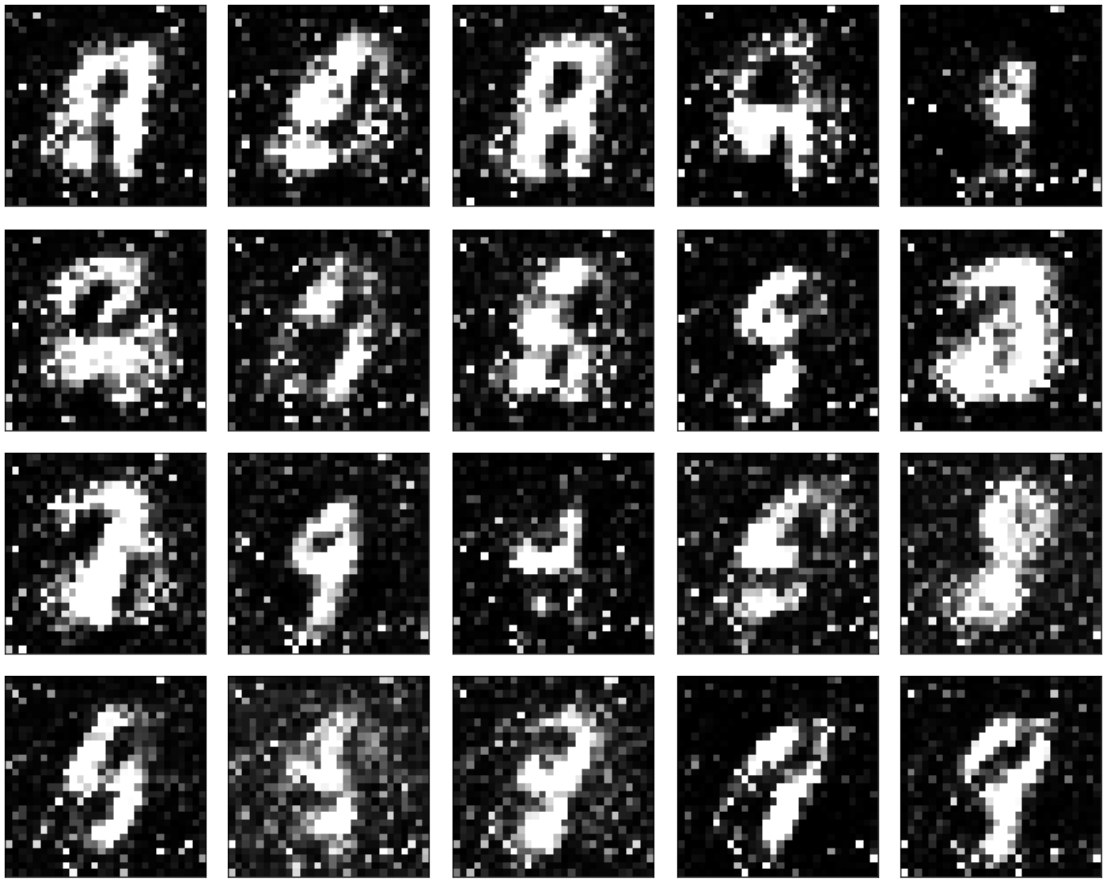
[ ]:
```
# hyperparameters
sample_size = 100 # latent sample size (i.e., 100 random numbers)
g_hidden_size = 128
d_hidden_size = 128
leaky_alpha = 0.01
g_learning_rate = 0.0001 # learning rate for the generator
d_learning_rate = 0.001 # learning rate for the discriminator
epochs = 10
batch_size = 64 # train batch size
eval_size = 16 # evaluate size
smooth = 0.1

# labels for the batch size and the test size
y_train_real, y_train_fake = make_labels(batch_size)
y_eval_real,  y_eval_fake  = make_labels(eval_size)

# create a GAN, a generator and a discriminator
gan, generator, discriminator = make_valina_GAN(
    sample_size,
    g_hidden_size,
    d_hidden_size,
    leaky_alpha,
    g_learning_rate,
    d_learning_rate)

losses = []
for e in range(epochs):
  for i in range(len(X_train_real)//batch_size):
    # real MNIST digit images
    X_batch_real = X_train_real[i*batch_size:(i+1)*batch_size]

    # latent samples and the generated digit images
    latent_samples = make_latent_samples(batch_size, sample_size)
    X_batch_fake = generator.predict_on_batch(latent_samples)

    # train the discriminator to detect real and fake images
    make_trainable(discriminator, True)
    discriminator.train_on_batch(X_batch_real, y_train_real * (1 - smooth))
    discriminator.train_on_batch(X_batch_fake, y_train_fake)

    # train the generator via GAN
```

```
    make_trainable(discriminator, False)
    gan.train_on_batch(latent_samples, y_train_real)

  # evaluate
  X_eval_real = X_test_real[np.random.choice(len(X_test_real), eval_size,␣
↪replace=False)]

  latent_samples = make_latent_samples(eval_size, sample_size)
  X_eval_fake = generator.predict_on_batch(latent_samples)

  d_loss  = discriminator.test_on_batch(X_eval_real, y_eval_real)
  d_loss += discriminator.test_on_batch(X_eval_fake, y_eval_fake)
  g_loss  = gan.test_on_batch(latent_samples, y_eval_real) # we want the fake␣
↪to be realistic!

  losses.append((d_loss, g_loss))

  print("Epoch: {:>3}/{} Discriminator Loss: {:>6.4f} Generator Loss: {:>6.4f}".
↪format(
      e+1, epochs, d_loss, g_loss))
```

```
Epoch:   1/10 Discriminator Loss: 0.2340 Generator Loss: 3.6429
Epoch:   2/10 Discriminator Loss: 0.1434 Generator Loss: 3.7076
Epoch:   3/10 Discriminator Loss: 0.2270 Generator Loss: 2.2539
Epoch:   4/10 Discriminator Loss: 0.3644 Generator Loss: 3.5880
Epoch:   5/10 Discriminator Loss: 0.1601 Generator Loss: 2.6983
Epoch:   6/10 Discriminator Loss: 0.2157 Generator Loss: 2.3662
Epoch:   7/10 Discriminator Loss: 0.2304 Generator Loss: 2.6454
Epoch:   8/10 Discriminator Loss: 0.5456 Generator Loss: 8.0517
Epoch:   9/10 Discriminator Loss: 0.4052 Generator Loss: 3.5438
Epoch:  10/10 Discriminator Loss: 0.3323 Generator Loss: 2.8379
```

[ ]:
```
losses = np.array(losses)

fig, ax = plt.subplots()
plt.plot(losses.T[0], label='Discriminator')
plt.plot(losses.T[1], label='Generator')
plt.title("Training Losses")
plt.legend()
plt.show()
```

[ ]:
```
latent_samples = make_latent_samples(20, sample_size)
generated_digits = generator.predict(latent_samples)

plt.figure(figsize=(10, 8))
for i in range(20):
    img = deprocess(generated_digits[i])
    plt.subplot(4, 5, i+1)
    plt.imshow(img, cmap='gray')
    plt.xticks([])
    plt.yticks([])
plt.tight_layout()
plt.show()
```



The results are not outstanding as we are using simple networks. Deep Convolutional GAN (aka DCGAN) would produce better results than this. So we then construct a DCGAN.

2 2. Deep Convolutional GAN (DCGAN) (SVHN)

We need a deeper and more powerful network than the **Valina GAN**. This is accomplished through using convolutional layers in the discriminator and generator. It's also necessary to use batch normalization to get the convolutional networks to train. The only real changes compared to Valina GAN are in the generator and discriminator,

[ ]:
```
from scipy.io import loadmat
import keras
from keras.layers import Dense, Activation, LeakyReLU, BatchNormalization
from keras.layers import Conv2D, Conv2DTranspose, Reshape, Flatten
from keras.models import Sequential
from keras.optimizers import Adam
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
%matplotlib inline
```

2.1 2.1 Load Data

[ ]:
```
train_data = loadmat('train_32x32.mat')
test_data  = loadmat('test_32x32.mat')
X_train, y_train = train_data['X'], train_data['y']
X_test,  y_test  = test_data['X'],  test_data['y']
# check the shape
X_train.shape
```

[ ]:
```
(32, 32, 3, 73257)
```

We roll the axis backward to make the shape to (records, image_height, image_width, channels).

[ ]:
```
# roll axis backward

X_train = np.rollaxis(X_train, 3)
X_test  = np.rollaxis(X_test, 3)
# Check the shape
X_train.shape
```

[ ]:
```
(73257, 32, 32, 3)
```

[ ]:
```
# Let's examine some sample images.

sample_images = X_train[np.random.choice(len(X_train), size=80, replace=False)]

plt.figure(figsize=(10, 8))
for i in range(80):
  plt.subplot(8, 10, i+1)
  plt.imshow(sample_images[i])
  plt.xticks([])
  plt.yticks([])
plt.tight_layout()
plt.show()
```

2.2 2.2 Preprocessing and Deprocessing

As usual, we need preprocessing and later deprocessing of the images.

As we will see later on, the generator is using tanh activation, for which we need to preprocess the image data into the range between -1 and 1.

[ ]:
```
def preprocess(x):
  return (x/255)*2-1

def deprocess(x):
  return np.uint8((x+1)/2*255) # make sure to use uint8 type otherwise the␣
↪image won't display properly
```

Apply the preprocessing on the train and test images (and they are the real images as oppose to the generated images).

[ ]:
```
X_train_real = preprocess(X_train)
X_test_real  = preprocess(X_test)
```

2.3 2.3 Generator

The generator takes a latent sample (100 randomly generated numbers) and produces a 32x32 color image that should look like one from the SVHN dataset.

generator network architecture details:

1. Although the size of the images used in the original paper is 64x64, the size of the images in SVHN is 32x32. As such, I used smaller networks.
2. The generator takes a latent sample which has 100 random numbers. It uses the fully connected layer to expand the dimension to 4×4×256 neurons so that it can be reshaped into 4x4 2D shape with 512 filters.
3. After that, each layer's height and width are doubled by the transpose convolution and the filters are halved. The last layer produces a 32x32 2D image with 3 channels.
4. The activation of the output layer is tanh which the discriminator expects;
5. Use Batch Normalization (BN) to stabilize learning by normalizing the input to each layer to have zero mean and unit variance;
6. Use Leaky ReLU in all layers of the discriminator. Unlike ReLU, instead of zeroing out all outputs when the input is less than zero, Leaky ReLU generates a small gradient equal to alpha x input.

[ ]:
```
def make_generator(input_size, leaky_alpha):
  # generates images in (32,32,3)
  return Sequential([
    Dense(4*4*512, input_shape=(input_size,)),
    Reshape(target_shape=(4, 4, 512)),                                  # 4,4,512
    BatchNormalization(),
    LeakyReLU(alpha=leaky_alpha),
    Conv2DTranspose(256, kernel_size=5, strides=2, padding='same'), # 8,8,256
    BatchNormalization(),
    LeakyReLU(alpha=leaky_alpha),
    Conv2DTranspose(128, kernel_size=5, strides=2, padding='same'), # 16,16,128
    BatchNormalization(),
    LeakyReLU(alpha=leaky_alpha),
    Conv2DTranspose(3, kernel_size=5, strides=2, padding='same',␣
↪activation='tanh')   # 32,32,3
  ])
```

2.4 2.4 Discriminator

The discriminator is a classifier to tell if the input image is real or fake.

It is a convolutional neural network that takes a 32x32 image with 3 channels. The values in the image is expected to be between -1 and 1.

The activation of the output layer is sigmoid and the discriminator outputs a probability of the image being real.

[ ]:
```
def make_discriminator(leaky_alpha):
  # classifies images in (32,32,3)
  return Sequential([
    Conv2D(64, kernel_size=5, strides=2, padding='same',␣
 ↪input_shape=(32,32,3)),  # 16,16,64
    LeakyReLU(alpha=leaky_alpha),
    Conv2D(128, kernel_size=5, strides=2, padding='same'),   # 8,8,128
    BatchNormalization(),
    LeakyReLU(alpha=leaky_alpha),
    Conv2D(256, kernel_size=5, strides=2, padding='same'),   # 4,4,256
    BatchNormalization(),
    LeakyReLU(alpha=leaky_alpha),
    Flatten(),
    Dense(1, activation='sigmoid')
  ])
```

2.5 2.5 DCGAN

We connect the generator and the discriminator to make a DCGAN.

The input to the DCGAN is a latent sample. The generator inside DCGAN produces an image which is fed into the discriminator inside the DCGAN. So, the output of DCGAN is the probability of the generated image being real.

[ ]:
```
# beta_1 is the exponential decay rate for the 1st moment estimates in Adam␣
 ↪optimizer
def make_DCGAN(sample_size,
        g_learning_rate,
        g_beta_1,
        d_learning_rate,
        d_beta_1,
        leaky_alpha):
  # clear any session data
  keras.backend.clear_session()

  # generator
  generator = make_generator(sample_size, leaky_alpha)

  # discriminator
  discriminator = make_discriminator(leaky_alpha)
  discriminator.compile(optimizer=Adam(lr=d_learning_rate, beta_1=d_beta_1),␣
 ↪loss='binary_crossentropy')

  # GAN
  gan = Sequential([generator, discriminator])
  gan.compile(optimizer=Adam(lr=g_learning_rate, beta_1=g_beta_1),␣
 ↪loss='binary_crossentropy')
```

```
  return gan, generator, discriminator
```

2.6 2.6 Training DCGAN

The below is a function to generate latent samples.

[ ]:
```
def make_latent_samples(n_samples, sample_size):
  #return np.random.uniform(-1, 1, size=(n_samples, sample_size))
  return np.random.normal(loc=0, scale=1, size=(n_samples, sample_size))

# The below is a function to set the discriminator to trainable or␣
 ↪non-trainable.
def make_trainable(model, trainable):
  for layer in model.layers:
    layer.trainable = trainable

# The below is a function to create a batch of labels.
def make_labels(size):
  return np.ones([size, 1]), np.zeros([size, 1])

# This is to show a epoch - loss chart.
def show_losses(losses):
  losses = np.array(losses)

  fig, ax = plt.subplots()
  plt.plot(losses.T[0], label='Discriminator')
  plt.plot(losses.T[1], label='Generator')
  plt.title("Validation Losses")
  plt.legend()
  plt.show()

# This is to show generated images.
def show_images(generated_images):
  n_images = len(generated_images)
  cols = 10
  rows = n_images//cols

  plt.figure(figsize=(10, 8))
  for i in range(n_images):
      img = deprocess(generated_images[i])
      ax = plt.subplot(rows, cols, i+1)
      plt.imshow(img)
      plt.xticks([])
      plt.yticks([])
  plt.tight_layout()
  plt.show()
```

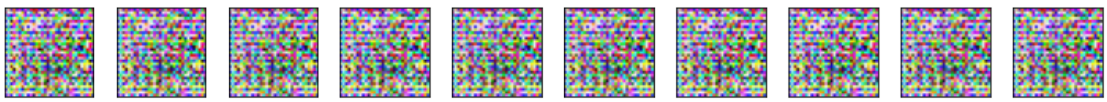
[ ]:
```
def train(
    g_learning_rate, # learning rate for the generator
    g_beta_1,        # the exponential decay rate for the 1st moment estimates␣
 ↪in Adam optimizer
    d_learning_rate, # learning rate for the discriminator
    d_beta_1,        # the exponential decay rate for the 1st moment estimates␣
 ↪in Adam optimizer
    leaky_alpha,
    smooth=0.1,
    sample_size=100, # latent sample size (i.e. 100 random numbers)
    epochs=15,
    batch_size= 128, # train batch size
    eval_size=16,    # evaluate size
    show_details=True):

    # labels for the batch size and the test size
    y_train_real, y_train_fake = make_labels(batch_size)
    y_eval_real,  y_eval_fake  = make_labels(eval_size)

    # create a GAN, a generator and a discriminator
    gan, generator, discriminator = make_DCGAN(
        sample_size,
        g_learning_rate,
        g_beta_1,
        d_learning_rate,
        d_beta_1,
        leaky_alpha)

    losses = []
    for e in range(epochs):
      for i in range(len(X_train_real)//batch_size):
        # real SVHN digit images
        X_batch_real = X_train_real[i*batch_size:(i+1)*batch_size]

        # latent samples and the generated digit images
        latent_samples = make_latent_samples(batch_size, sample_size)
        X_batch_fake = generator.predict_on_batch(latent_samples)

        # train the discriminator to detect real and fake images
        make_trainable(discriminator, True)
        discriminator.train_on_batch(X_batch_real, y_train_real * (1 - smooth))
        discriminator.train_on_batch(X_batch_fake, y_train_fake)

        # train the generator via GAN
        make_trainable(discriminator, False)
        gan.train_on_batch(latent_samples, y_train_real)
```

```
    # evaluate
    X_eval_real = X_test_real[np.random.choice(len(X_test_real), eval_size,␣
↪replace=False)]

    latent_samples = make_latent_samples(eval_size, sample_size)
    X_eval_fake = generator.predict_on_batch(latent_samples)

    d_loss  = discriminator.test_on_batch(X_eval_real, y_eval_real)
    d_loss += discriminator.test_on_batch(X_eval_fake, y_eval_fake)
    g_loss  = gan.test_on_batch(latent_samples, y_eval_real) # we want the␣
↪fake to be realistic!

    losses.append((d_loss, g_loss))

    print("Epoch: {:>3}/{} Discriminator Loss: {:>7.4f}  Generator Loss: {:>7.
↪4f}".format(
        e+1, epochs, d_loss, g_loss))

    # show the generated images
    if (e+1) % 5 == 0:
      show_images(X_eval_fake[:10])

  if show_details:
    show_losses(losses)
    show_images(generator.predict(make_latent_samples(80, sample_size)))
  return generator
```

2.6.1 Traning 1

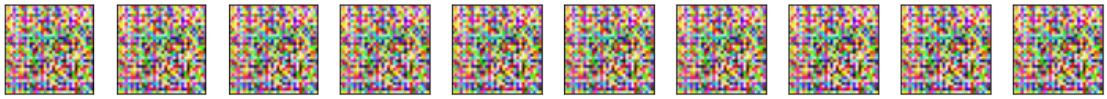
[ ]:
```
train(g_learning_rate=0.0002, g_beta_1=0.5, d_learning_rate=0.0002, d_beta_1=0.
↪5, leaky_alpha=0.2)
```

```
Epoch:   1/15 Discriminator Loss:  3.1320  Generator Loss:  1.1729
Epoch:   2/15 Discriminator Loss:  5.3761  Generator Loss:  0.5561
Epoch:   3/15 Discriminator Loss:  4.0039  Generator Loss:  0.2028
Epoch:   4/15 Discriminator Loss:  1.7158  Generator Loss:  1.5672
Epoch:   5/15 Discriminator Loss:  5.2198  Generator Loss:  1.1323
```

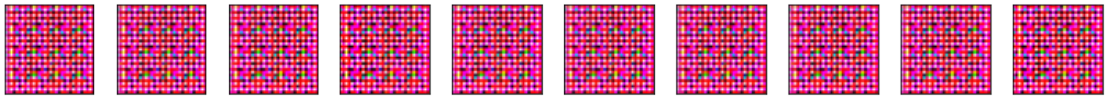


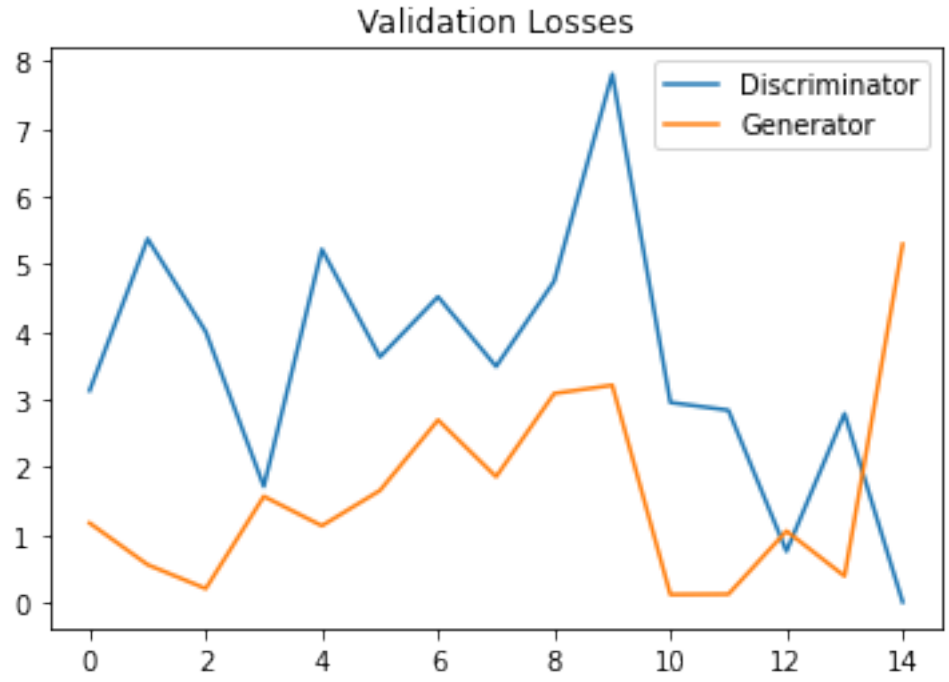
```
Epoch:   6/15 Discriminator Loss:  3.6292  Generator Loss:  1.6544
Epoch:   7/15 Discriminator Loss:  4.5192  Generator Loss:  2.6973
Epoch:   8/15 Discriminator Loss:  3.4882  Generator Loss:  1.8580
```

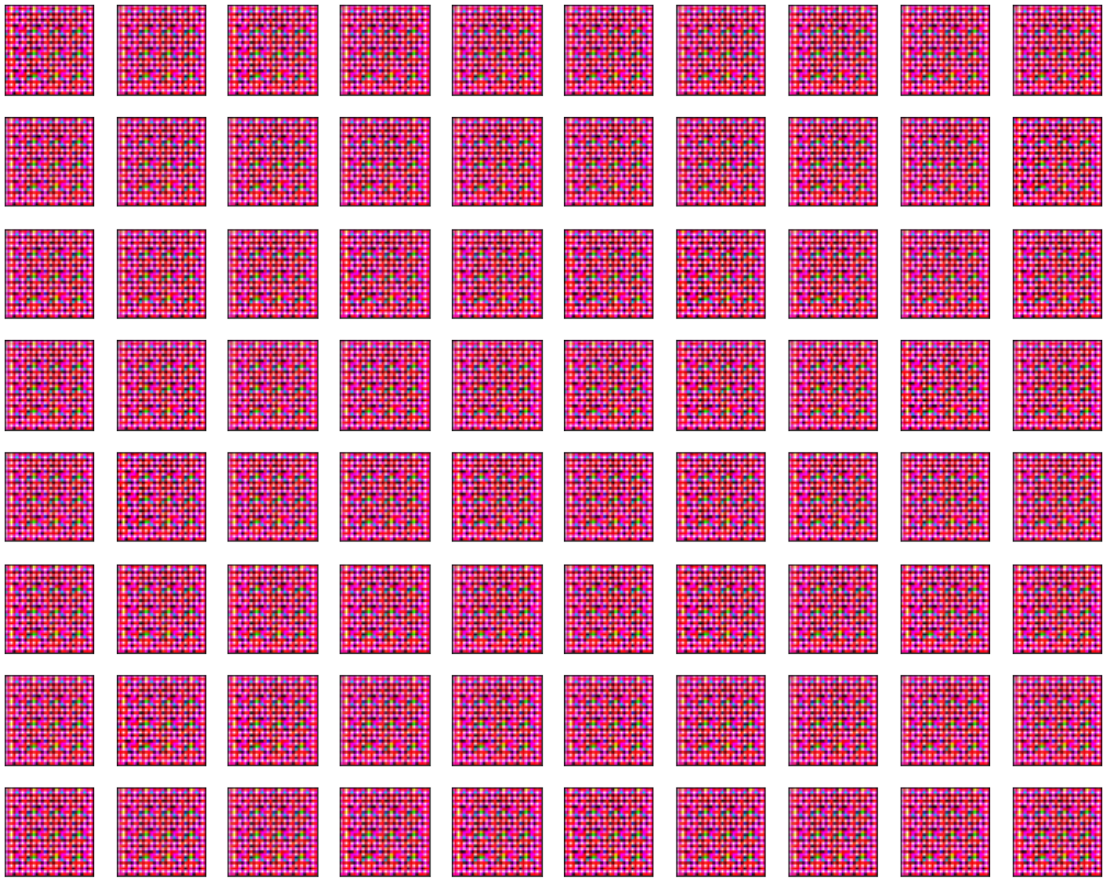
```
Epoch:   9/15 Discriminator Loss:  4.7503  Generator Loss:  3.0886
Epoch:  10/15 Discriminator Loss:  7.8125  Generator Loss:  3.2107
```



```
Epoch:  11/15 Discriminator Loss:  2.9559  Generator Loss:  0.1157
Epoch:  12/15 Discriminator Loss:  2.8413  Generator Loss:  0.1202
Epoch:  13/15 Discriminator Loss:  0.7558  Generator Loss:  1.0543
Epoch:  14/15 Discriminator Loss:  2.7915  Generator Loss:  0.3881
Epoch:  15/15 Discriminator Loss:  0.0051  Generator Loss:  5.2959
```







[ ]: <tensorflow.python.keras.engine.sequential.Sequential at 0x7f954d378e80>

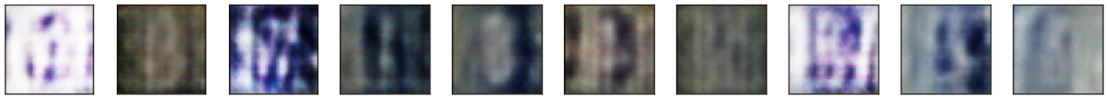
As we can see: 1. The generated images are garbages; 2. The Discriminator Loss and Generator Loss are flactuating.

So we need to change some hyperparameters. In the **Training 2**, I reduce the generator learning rate and increase the discriminator learning rate.

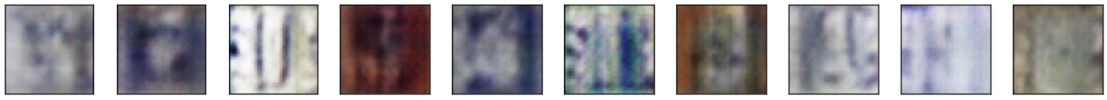
2.6.2 Training 2

[ ]:
```
train(g_learning_rate=0.0001, g_beta_1=0.5, d_learning_rate=0.01,d_beta_1=0.
↪5,leaky_alpha=0.2,epochs=30)
```

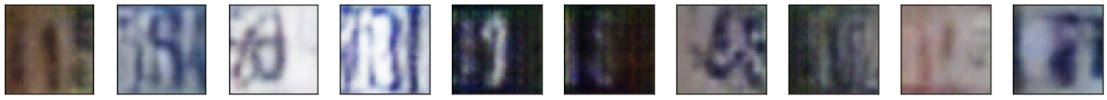
```
Epoch:   1/30 Discriminator Loss:  1.2328  Generator Loss:  3.3487
Epoch:   2/30 Discriminator Loss:  0.5246  Generator Loss:  1.7293
Epoch:   3/30 Discriminator Loss:  1.0402  Generator Loss:  1.6299
Epoch:   4/30 Discriminator Loss:  1.5336  Generator Loss:  1.0431
Epoch:   5/30 Discriminator Loss:  1.1046  Generator Loss:  1.1122
```



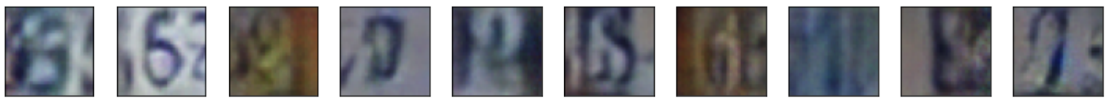
```
Epoch:   6/30 Discriminator Loss:  1.3320  Generator Loss:  0.7896
Epoch:   7/30 Discriminator Loss:  1.0783  Generator Loss:  1.0096
Epoch:   8/30 Discriminator Loss:  1.4289  Generator Loss:  1.9215
Epoch:   9/30 Discriminator Loss:  1.0953  Generator Loss:  1.5011
Epoch:  10/30 Discriminator Loss:  0.9044  Generator Loss:  1.4669
```



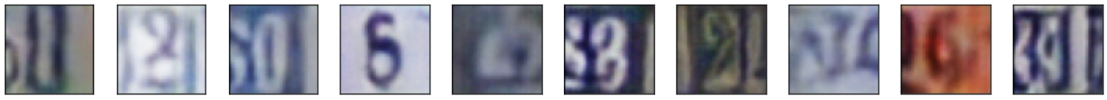
```
Epoch:  11/30 Discriminator Loss:  1.2514  Generator Loss:  1.2388
Epoch:  12/30 Discriminator Loss:  1.1537  Generator Loss:  1.2787
Epoch:  13/30 Discriminator Loss:  1.2757  Generator Loss:  2.8621
Epoch:  14/30 Discriminator Loss:  1.1984  Generator Loss:  2.5152
Epoch:  15/30 Discriminator Loss:  1.0648  Generator Loss:  1.8739
```



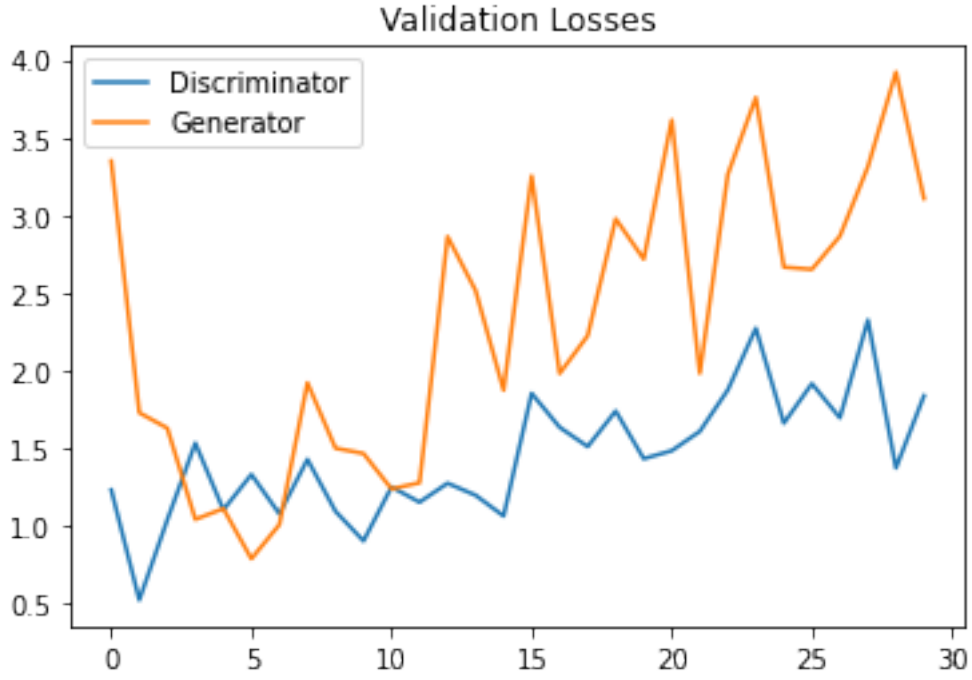
```
Epoch:  16/30 Discriminator Loss:  1.8536  Generator Loss:  3.2525
Epoch:  17/30 Discriminator Loss:  1.6353  Generator Loss:  1.9819
Epoch:  18/30 Discriminator Loss:  1.5115  Generator Loss:  2.2249
Epoch:  19/30 Discriminator Loss:  1.7402  Generator Loss:  2.9766
Epoch:  20/30 Discriminator Loss:  1.4333  Generator Loss:  2.7174
```



```
Epoch:  21/30 Discriminator Loss:  1.4849  Generator Loss:  3.6122
Epoch:  22/30 Discriminator Loss:  1.6092  Generator Loss:  1.9829
Epoch:  23/30 Discriminator Loss:  1.8758  Generator Loss:  3.2621
Epoch:  24/30 Discriminator Loss:  2.2722  Generator Loss:  3.7564
Epoch:  25/30 Discriminator Loss:  1.6618  Generator Loss:  2.6658
```



```
Epoch:  26/30 Discriminator Loss:  1.9160  Generator Loss:  2.6513
Epoch:  27/30 Discriminator Loss:  1.6966  Generator Loss:  2.8653
Epoch:  28/30 Discriminator Loss:  2.3255  Generator Loss:  3.3123
Epoch:  29/30 Discriminator Loss:  1.3739  Generator Loss:  3.9202
Epoch:  30/30 Discriminator Loss:  1.8384  Generator Loss:  3.1068
```



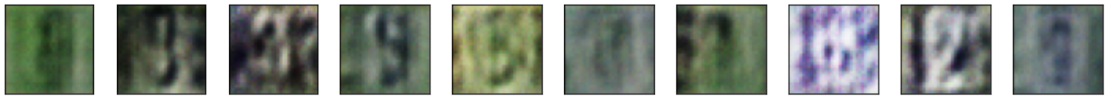
[ ]: <tensorflow.python.keras.engine.sequential.Sequential at 0x7f94ec347668>

- The generated images are much better than those generated in **Training 1**;
- Both Discriminator Loss and Generator Loss are more stable;
- More hyperparameters should be tuned if we need

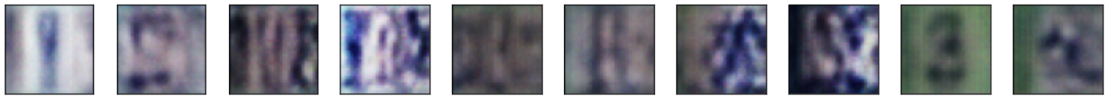
Then I tried some more sets of parameters, the results are not improved muc. Therefore, I choose the set of parameters in **Training 2** as my best and final choice.

[ ]:
```
train(g_learning_rate=0.00005, g_beta_1=0.5, d_learning_rate=0.02,d_beta_1=0.
↪5,leaky_alpha=0.2,epochs=30)
```

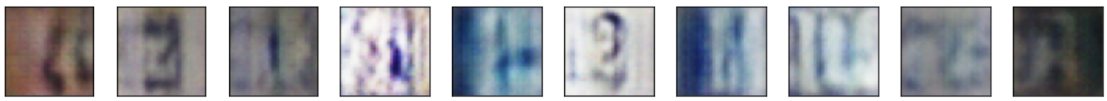
```
Epoch:   1/30 Discriminator Loss:  1.2770  Generator Loss:  1.2363
Epoch:   2/30 Discriminator Loss:  1.1612  Generator Loss:  0.6235
Epoch:   3/30 Discriminator Loss:  1.1418  Generator Loss:  2.7275
Epoch:   4/30 Discriminator Loss:  1.6934  Generator Loss:  2.7327
Epoch:   5/30 Discriminator Loss:  1.0131  Generator Loss:  1.1130
```



```
Epoch:   6/30 Discriminator Loss:  0.9883  Generator Loss:  0.9592
Epoch:   7/30 Discriminator Loss:  1.4496  Generator Loss:  5.7391
Epoch:   8/30 Discriminator Loss:  0.7999  Generator Loss:  1.4972
Epoch:   9/30 Discriminator Loss:  0.9019  Generator Loss:  1.2910
Epoch:  10/30 Discriminator Loss:  0.9362  Generator Loss:  1.1241
```

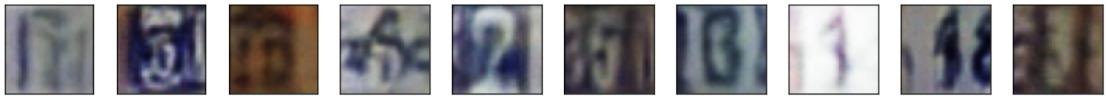


```
Epoch:  11/30 Discriminator Loss:  1.2916  Generator Loss:  1.8772
Epoch:  12/30 Discriminator Loss:  1.1768  Generator Loss:  2.8164
Epoch:  13/30 Discriminator Loss:  1.2232  Generator Loss:  1.5949
Epoch:  14/30 Discriminator Loss:  1.3712  Generator Loss:  2.1175
Epoch:  15/30 Discriminator Loss:  0.8710  Generator Loss:  2.7578
```

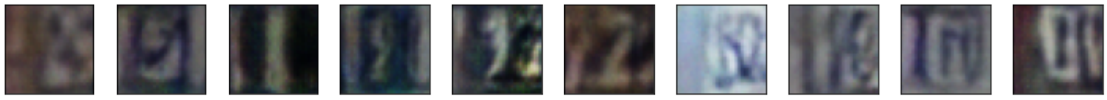


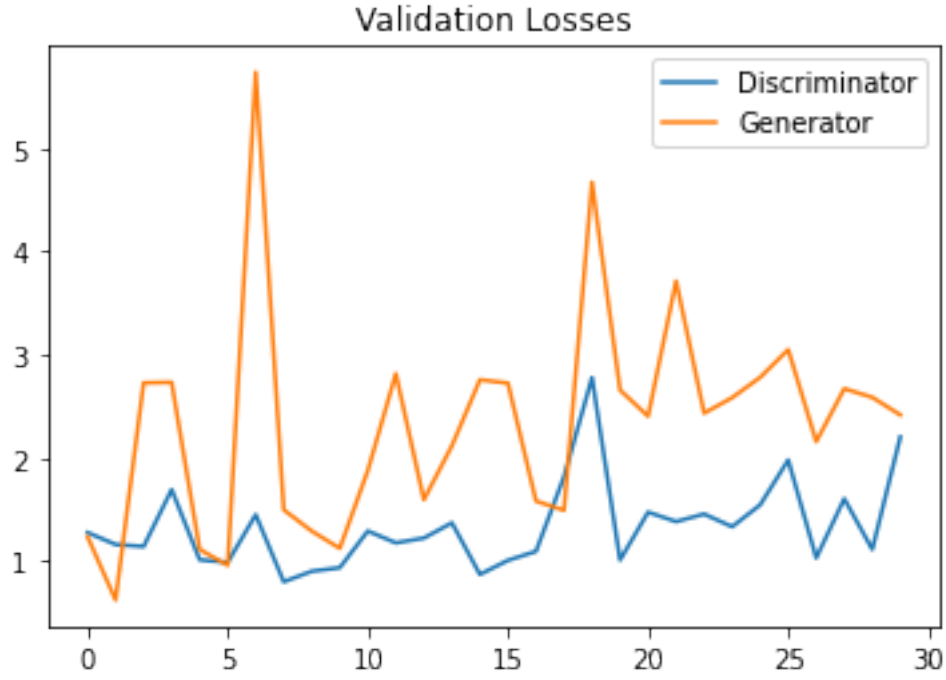
```
Epoch:  16/30 Discriminator Loss:  1.0078  Generator Loss:  2.7250
Epoch:  17/30 Discriminator Loss:  1.0945  Generator Loss:  1.5822
Epoch:  18/30 Discriminator Loss:  1.8182  Generator Loss:  1.4925
Epoch:  19/30 Discriminator Loss:  2.7788  Generator Loss:  4.6726
Epoch:  20/30 Discriminator Loss:  1.0106  Generator Loss:  2.6580
```

```
Epoch:  21/30 Discriminator Loss:  1.4743  Generator Loss:  2.4036
Epoch:  22/30 Discriminator Loss:  1.3832  Generator Loss:  3.7150
Epoch:  23/30 Discriminator Loss:  1.4561  Generator Loss:  2.4335
Epoch:  24/30 Discriminator Loss:  1.3333  Generator Loss:  2.5840
Epoch:  25/30 Discriminator Loss:  1.5454  Generator Loss:  2.7812
```



```
Epoch:  26/30 Discriminator Loss:  1.9799  Generator Loss:  3.0500
Epoch:  27/30 Discriminator Loss:  1.0310  Generator Loss:  2.1579
Epoch:  28/30 Discriminator Loss:  1.6047  Generator Loss:  2.6711
Epoch:  29/30 Discriminator Loss:  1.1138  Generator Loss:  2.5873
Epoch:  30/30 Discriminator Loss:  2.2034  Generator Loss:  2.4170
```

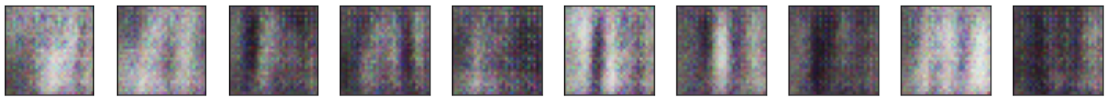




[ ]: <tensorflow.python.keras.engine.sequential.Sequential at 0x7f94ec67eda0>

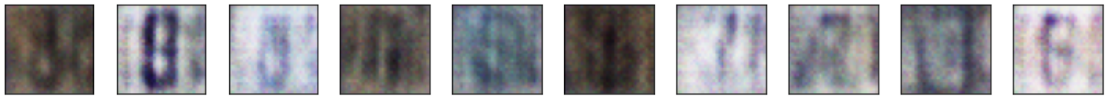
[ ]:
```
train(g_learning_rate=0.00001, g_beta_1=0.5, d_learning_rate=0.05,d_beta_1=0.
↪5,leaky_alpha=0.2,epochs=30)
```

```
Epoch:   1/30 Discriminator Loss:  0.3767  Generator Loss:  2.5520
Epoch:   2/30 Discriminator Loss:  2.6239  Generator Loss:  1.6784
Epoch:   3/30 Discriminator Loss:  2.7168  Generator Loss:  0.5637
Epoch:   4/30 Discriminator Loss:  2.4379  Generator Loss:  3.6998
Epoch:   5/30 Discriminator Loss:  0.8477  Generator Loss:  1.5089
```

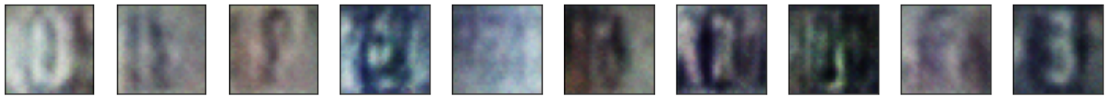


```
Epoch:   6/30 Discriminator Loss:  0.6542  Generator Loss:  2.2797
Epoch:   7/30 Discriminator Loss:  0.6824  Generator Loss:  2.8205
```

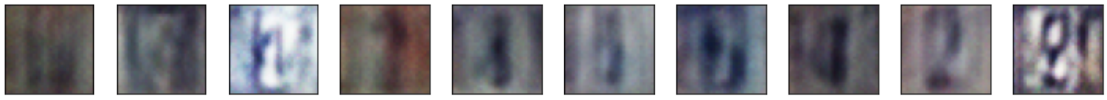
```
Epoch:   8/30 Discriminator Loss:  2.0235  Generator Loss:  2.1989
Epoch:   9/30 Discriminator Loss:  2.5013  Generator Loss:  4.2709
Epoch:  10/30 Discriminator Loss:  2.8175  Generator Loss:  6.1106
```



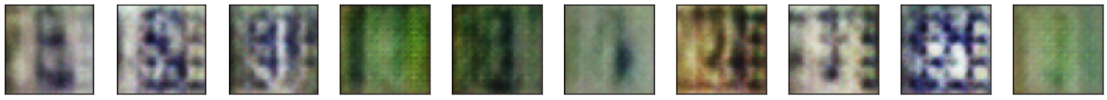
```
Epoch:  11/30 Discriminator Loss:  1.7955  Generator Loss:  6.0395
Epoch:  12/30 Discriminator Loss:  7.9617  Generator Loss: 33.5875
Epoch:  13/30 Discriminator Loss:  0.8088  Generator Loss:  3.6637
Epoch:  14/30 Discriminator Loss:  6.6866  Generator Loss: 19.4845
Epoch:  15/30 Discriminator Loss:  0.6776  Generator Loss:  2.9917
```



```
Epoch:  16/30 Discriminator Loss:  0.5247  Generator Loss:  2.2894
Epoch:  17/30 Discriminator Loss:  0.9597  Generator Loss:  2.0207
Epoch:  18/30 Discriminator Loss:  0.9406  Generator Loss:  2.4064
Epoch:  19/30 Discriminator Loss:  1.4748  Generator Loss:  2.4886
Epoch:  20/30 Discriminator Loss:  0.8626  Generator Loss:  1.0927
```

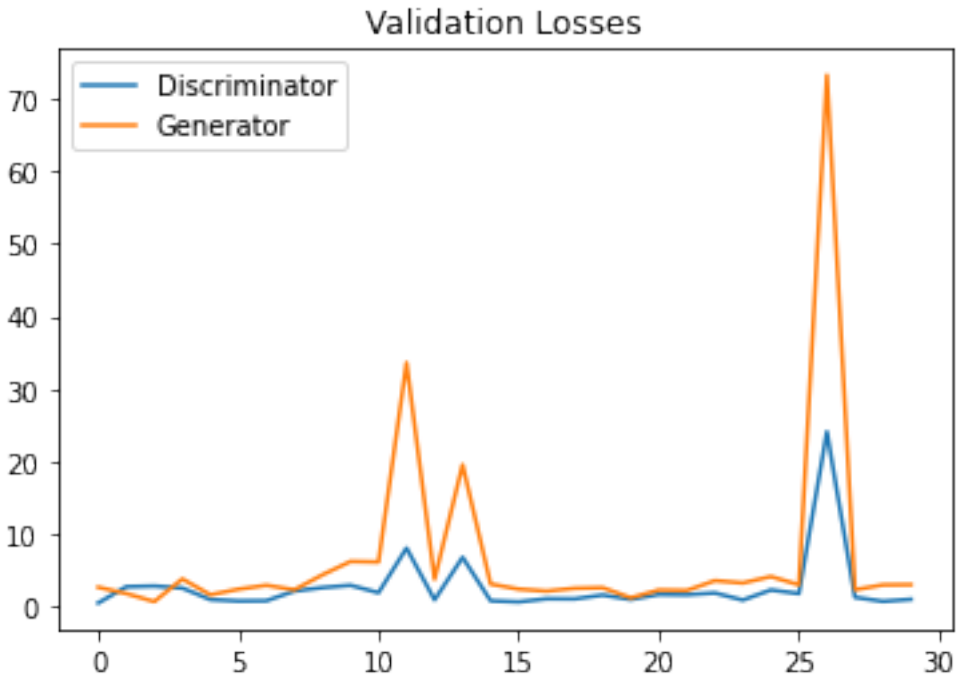


```
Epoch:  21/30 Discriminator Loss:  1.5593  Generator Loss:  2.1576
Epoch:  22/30 Discriminator Loss:  1.5161  Generator Loss:  2.1021
Epoch:  23/30 Discriminator Loss:  1.7569  Generator Loss:  3.4592
Epoch:  24/30 Discriminator Loss:  0.7784  Generator Loss:  3.1416
Epoch:  25/30 Discriminator Loss:  2.1753  Generator Loss:  4.0277
```



```
Epoch:  26/30 Discriminator Loss:  1.6717  Generator Loss:  2.8912
```

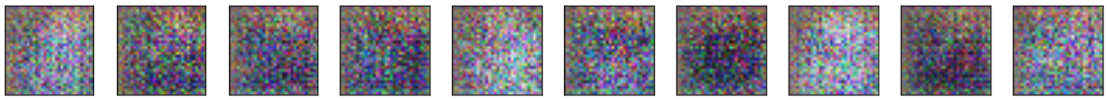
```
Epoch:  27/30 Discriminator Loss: 24.0483  Generator Loss: 73.3400
Epoch:  28/30 Discriminator Loss:  1.1593  Generator Loss:  2.2006
Epoch:  29/30 Discriminator Loss:  0.5993  Generator Loss:  2.8796
Epoch:  30/30 Discriminator Loss:  0.8830  Generator Loss:  2.9070
```



[ ]: <tensorflow.python.keras.engine.sequential.Sequential at 0x7f94ec085da0>

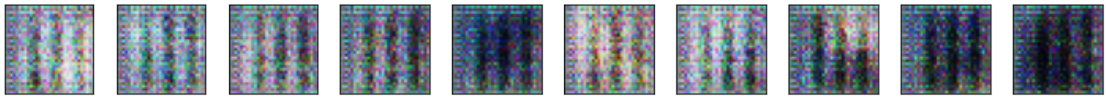
[ ]:
```
train(g_learning_rate=0.000001, g_beta_1=0.5, d_learning_rate=0.005,d_beta_1=0.
↪5,leaky_alpha=0.2,epochs=30)
```

```
Epoch:   1/30 Discriminator Loss:  0.6745  Generator Loss:  2.3839
Epoch:   2/30 Discriminator Loss:  1.1585  Generator Loss:  1.7885
Epoch:   3/30 Discriminator Loss:  0.4340  Generator Loss:  3.1496
Epoch:   4/30 Discriminator Loss:  0.8113  Generator Loss:  3.2219
Epoch:   5/30 Discriminator Loss:  1.8869  Generator Loss:  1.4441
```

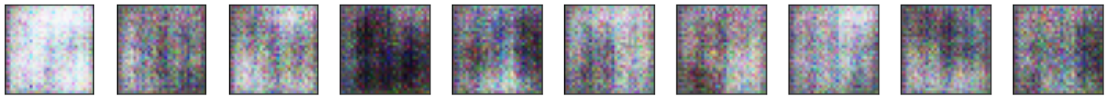


```
Epoch:   6/30 Discriminator Loss:  0.5010  Generator Loss:  3.4731
Epoch:   7/30 Discriminator Loss:  0.7582  Generator Loss:  2.6403
```

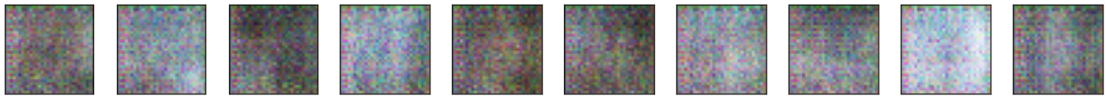
```
Epoch:   8/30 Discriminator Loss:  0.5535  Generator Loss:  3.2738
Epoch:   9/30 Discriminator Loss:  1.7672  Generator Loss:  1.7980
Epoch:  10/30 Discriminator Loss:  1.2257  Generator Loss:  3.1849
```



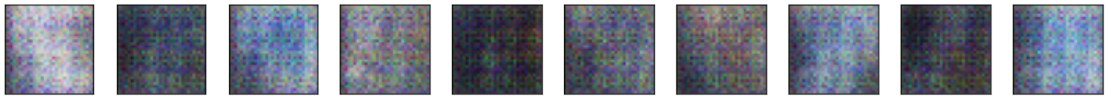
```
Epoch:  11/30 Discriminator Loss:  0.9833  Generator Loss:  5.9734
Epoch:  12/30 Discriminator Loss:  0.6615  Generator Loss:  5.9615
Epoch:  13/30 Discriminator Loss:  0.8769  Generator Loss:  4.2006
Epoch:  14/30 Discriminator Loss:  0.5106  Generator Loss:  1.8936
Epoch:  15/30 Discriminator Loss:  0.7352  Generator Loss:  1.5525
```



```
Epoch:  16/30 Discriminator Loss:  0.8532  Generator Loss:  4.5858
Epoch:  17/30 Discriminator Loss:  1.1971  Generator Loss:  2.0261
Epoch:  18/30 Discriminator Loss:  2.5867  Generator Loss:  3.6982
Epoch:  19/30 Discriminator Loss:  1.6283  Generator Loss:  2.4291
Epoch:  20/30 Discriminator Loss:  0.2203  Generator Loss:  5.9381
```

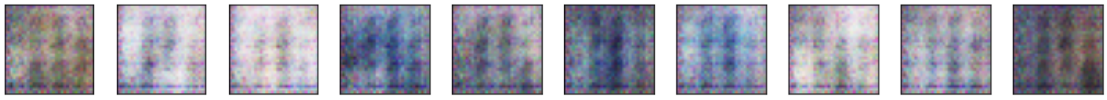


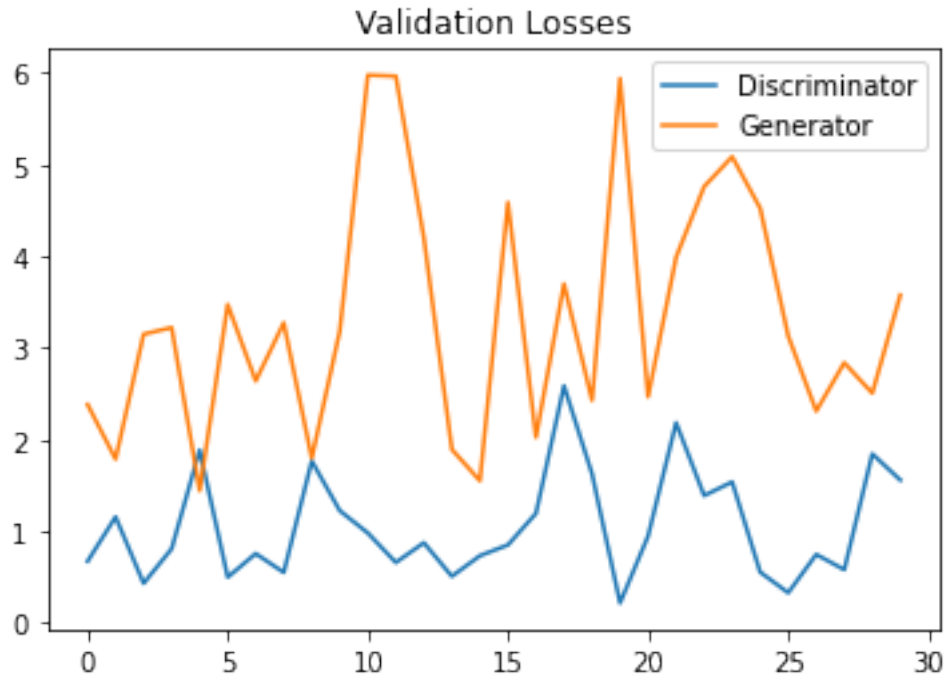
```
Epoch:  21/30 Discriminator Loss:  0.9498  Generator Loss:  2.4681
Epoch:  22/30 Discriminator Loss:  2.1845  Generator Loss:  3.9888
Epoch:  23/30 Discriminator Loss:  1.3904  Generator Loss:  4.7585
Epoch:  24/30 Discriminator Loss:  1.5392  Generator Loss:  5.0845
Epoch:  25/30 Discriminator Loss:  0.5566  Generator Loss:  4.5224
```

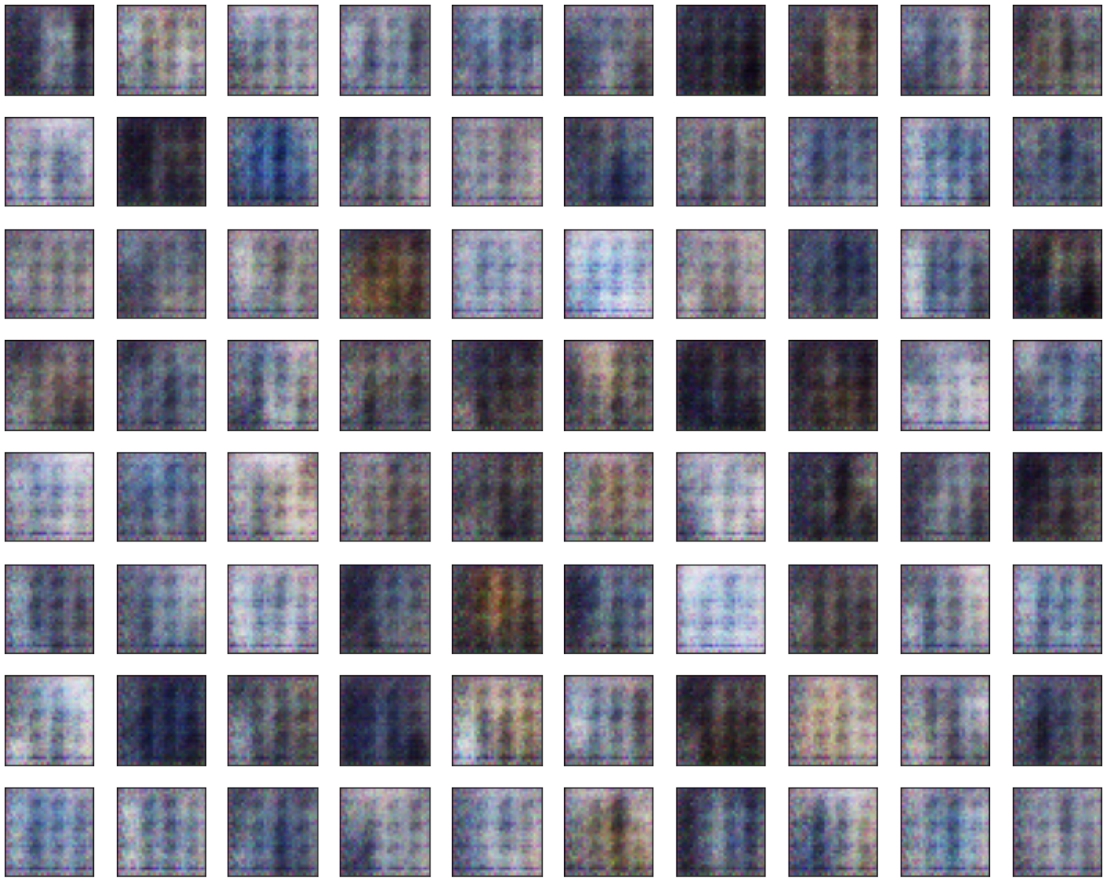


```
Epoch:  26/30 Discriminator Loss:  0.3316  Generator Loss:  3.1305
```

```
Epoch:  27/30 Discriminator Loss:  0.7492  Generator Loss:  2.3119
Epoch:  28/30 Discriminator Loss:  0.5829  Generator Loss:  2.8387
Epoch:  29/30 Discriminator Loss:  1.8454  Generator Loss:  2.5069
Epoch:  30/30 Discriminator Loss:  1.5596  Generator Loss:  3.5765
```







[ ]: <tensorflow.python.keras.engine.sequential.Sequential at 0x7f946b7ec5c0>

2.7 2.7 Conclusion

1. Overall, the training 2 performed the best in terms of realness of the generated images;
2. DCGANs are very sensitive to hyperparameters, More hyperparameter tuning should be experimented if we want to improve further;
3. Limited by computing power and speed, we couldn't try some more sets of parameters, which leads to a not outstanding result of our DCGAN.

[ ]:

WGAN_MNIST

December 21, 2020

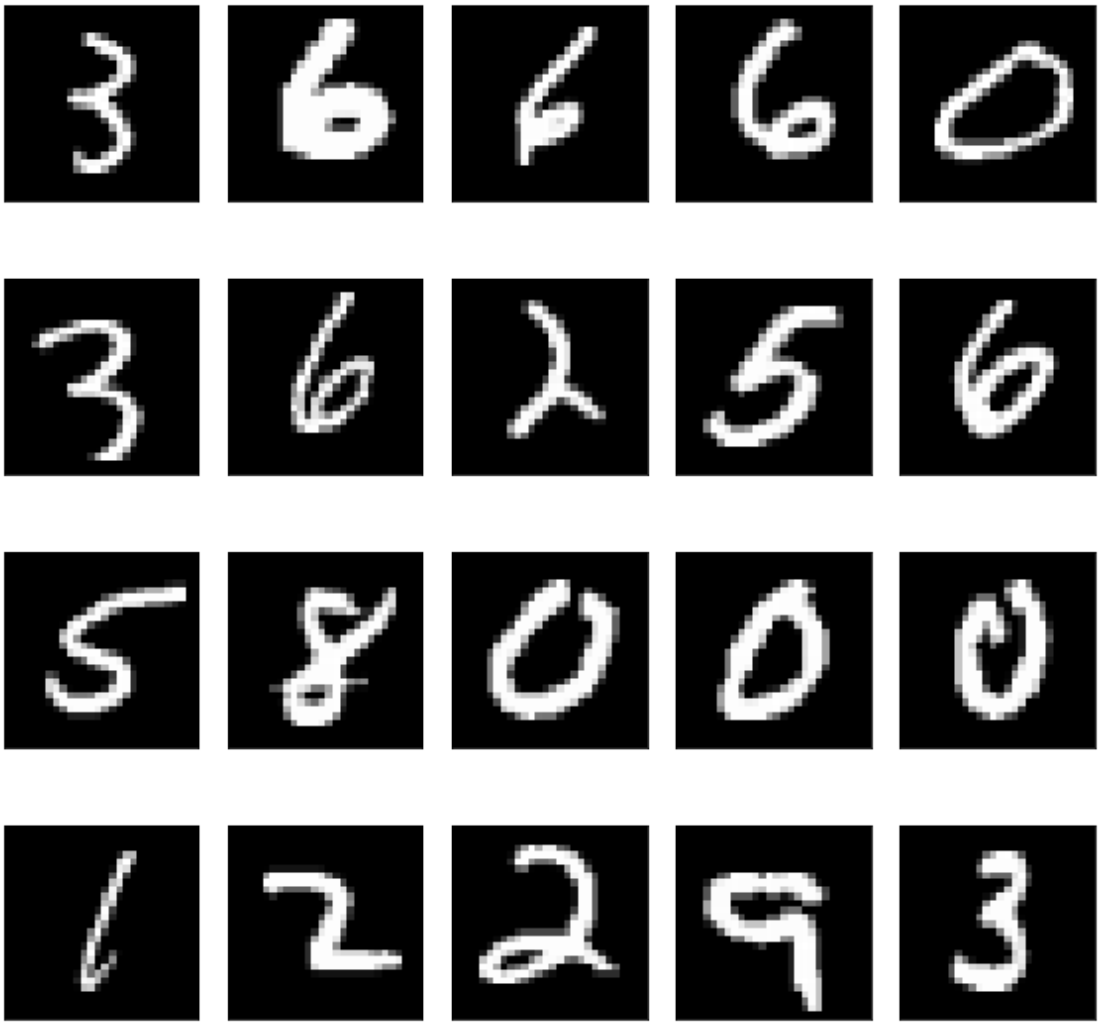
[1]:
```
import numpy as np
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.layers import Input, Conv2D, Dense, BatchNormalization,␣
↪LeakyReLU, Reshape, Conv2DTranspose, Dropout, Flatten, AveragePooling2D,␣
↪ReLU, UpSampling2D
from tensorflow.keras import backend
from tensorflow.keras.constraints import Constraint
```

0.1 Load Data

[2]:
```
(x_train, y_train), (_, _) = mnist.load_data()
print('x_train shape:', x_train.shape)
print('y_train shape:', y_train.shape)
```

```
x_train shape: (60000, 28, 28)
y_train shape: (60000,)
```

[3]:
```
### Let's examine some sample images.
np.random.seed(seed = 0)
sample_images = x_train[np.random.choice(len(x_train), size=20, replace=False)]

plt.figure(figsize=(8, 8))
for i in range(20):
  plt.subplot(4, 5, i+1)
  plt.imshow(sample_images[i], cmap = 'gray')
  plt.xticks([])
  plt.yticks([])
plt.tight_layout()
plt.show()
```



0.2 Data Reshape

In this part, I first normalize each image array, to $[-1, 1]$, since the activation function we use in Generator is tanh. Then I reshape each image, since I plan to use the simple network for both Generator and Discriminator, and since the images are all have only one color channel (28, 28), I reshape each image to $(28 \times 28,)$.

[4]:
```
### preprocess
def preprocess(x):
  x = x.reshape(x.shape[0], x.shape[1] * x.shape[2])
  x = (x.astype(np.float32) - 127.5) /  127.5 ## normalize the image to -1 and 1

  return x

x_train = preprocess(x_train)
```

```
print('x_train shape:', x_train.shape)
```

```
x_train shape: (60000, 784)
```

0.3 Wasserstein Loss

This is a self defined loss function, I will explain it in detail in the following train() function.

[5]:
```
def wasserstein_loss(y_true, y_pred):
        return tf.reduce_mean(y_true * y_pred)
```

0.4 Generator

Architecture: Input(100,) -> Dense(128) -> LeakyReLU(0.01) -> Dense(784) -> Tanh

[6]:
```
generator = tf.keras.Sequential([

                                Dense(128, input_shape = (100, )),
                                LeakyReLU(alpha = 0.01),
                                Dense(784, activation = 'tanh')

                                #Dense(7*7*512, input_shape = (100, )),
                                #BatchNormalization(),
                                #ReLU(),

                                #Reshape((7, 7, 512)),

                                #Conv2DTranspose(filters = 256, kernel_size = (3,␣
 ↪3), strides = (2, 2), padding = 'same', use_bias = False),
                                #BatchNormalization(),
                                #ReLU(),

                                #Conv2DTranspose(filters = 128, kernel_size = (4,␣
 ↪4), strides = (2, 2), padding = 'same', use_bias = False),
                                #BatchNormalization(),
                                #ReLU(),

                                #Conv2DTranspose(filters = 1, kernel_size = (4,␣
 ↪4), strides = (1, 1), padding = 'same', use_bias = False, activation =␣
 ↪'tanh'),
                                #Conv2D(filters = 1, kernel_size = (7, 7), padding␣
 ↪= 'same', activation = 'tanh')
])
#generator.compile(optimizer=tf.keras.optimizers.RMSprop(0.00005), loss =␣
 ↪wasserstein_loss)
```

[7]:
```
generator.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 128)               12928
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 784)               101136
=================================================================
Total params: 114,064
Trainable params: 114,064
Non-trainable params: 0
_________________________________________________________________
```

0.5 Discriminator

Input(784,) -> Dense(128) -> LeakyReLU(0.01) -> Dense(1) -> Linear

[8]:
```
discriminator = tf.keras.Sequential([
                                     Dense(128, input_shape = (784, )),
                                     LeakyReLU(alpha = 0.01),
                                     Dense(1, activation='linear')
                                     #Conv2D(filters = 512, kernel_size = (3,␣
↪3), strides = (2, 2), padding = 'same', input_shape = (28, 28, 1), use_bias␣
↪= True, kernel_constraint=const),
                                     #BatchNormalization(),
                                     #LeakyReLU(alpha = 0.2),
                                     #Dropout(0.3),

                                     #Conv2D(filters = 256, kernel_size = (3,␣
↪3), strides = (2, 2), padding = 'same', use_bias = True, kernel_constraint =␣
↪const),
                                     #BatchNormalization(),
                                     #LeakyReLU(alpha = 0.2),
                                     #Dropout(0.3),
                                     #Conv2D(filters = 128, kernel_size = (3,␣
↪3), strides = (2, 2), padding = 'same', use_bias = True,␣
↪kernel_constraint=const),
                                     #BatchNormalization(),
                                     #LeakyReLU(alpha = 0.2),
                                     #AveragePooling2D(pool_size = (4, 4)),

                                     #Flatten(),
```

```
                                    #Dense(1, activation = 'linear')
])
discriminator.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate = 0.
↪00005), loss = wasserstein_loss)
```

[9]:
```
discriminator.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 128)               100480
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 129
=================================================================
Total params: 100,609
Trainable params: 100,609
Non-trainable params: 0
_________________________________________________________________
```

0.6 Combine 2 Networks

[10]:
```
def w_gan(generator, discriminator):
  discriminator.trainable = False
  model = tf.keras.Sequential()
  model.add(generator)
  model.add(discriminator)

  return model

wgan = w_gan(generator, discriminator)
wgan.compile(optimizer = tf.keras.optimizers.RMSprop(learning_rate = 0.00005),
↪loss = wasserstein_loss)
```

0.7 Train

The formula of loss function:

$$DiscriminatorLoss = E_{z\sim p_z(z)}(D(G(z))) - E_{x\sim p_{data}(x)}(D(x))$$

$$GeneratorLoss = -E_{z\sim p_z(z)}(D(G(z)))$$

In the train() functin, I train the networks with MNIST dataset. To be specific:

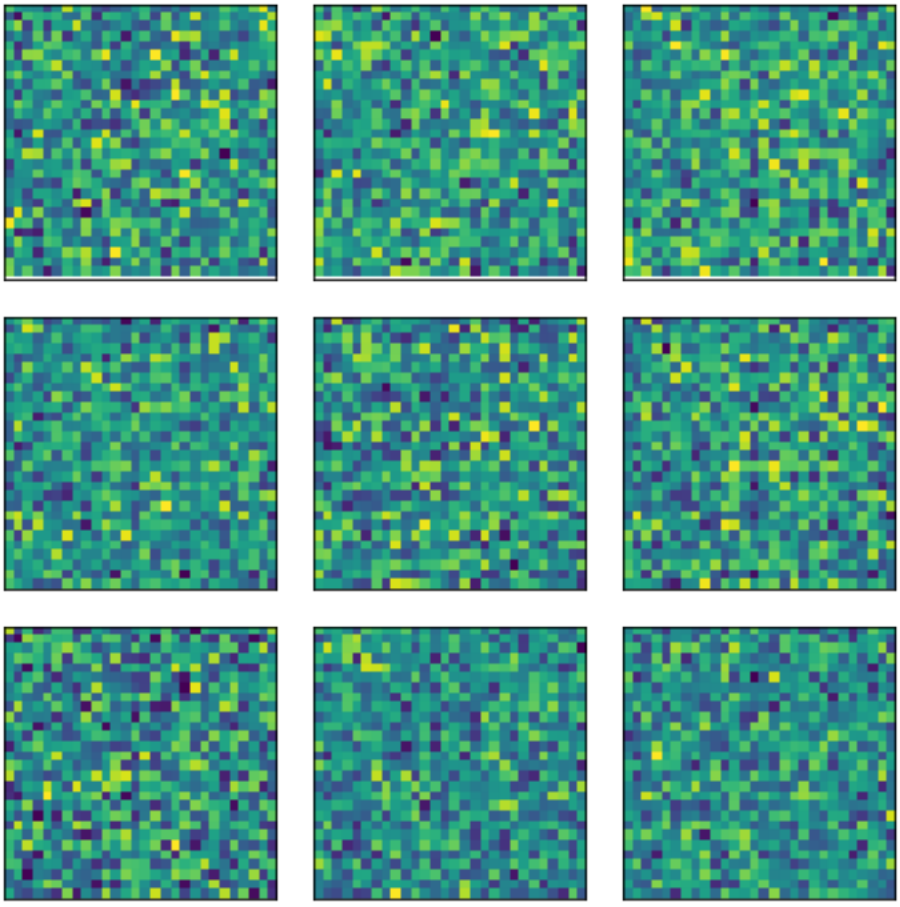
- in the discriminator loss, the coefficient of true images are **-1**, so when I set the true label of each true images, I set it as $y_{true} = -1$; the coefficient of fake images are **1**, so when I set the fake label of each fake images, I set it as $y_{fake} = 1$.
- now we come to the Wasserstein Loss I defined above: mean$y_{pred} * y_{true}$. When I multiply the true labels (all -1) with the output of discriminator(true images), we can get the second item of discriminator loss $discriminator_{trueimages} * y_{true}$;
- Similarly, when I multiply the fake labels with the output of discriminator(fake images), we can get the first item of discriminator loss $discriminator_{fakeimages} * y_{fake}$;
- Similarly, when I multiply the output of WGAN model and the true labels, I can get the generator loss. $WGAN_{fakeimages} * y_{true}$
- Also, I have included the part in the WGAN algorithm that limit the value of discriminator weights between **-0.01** and **0.01**
- During the training process, I also show the D loss and G loss every 500 epochs and show the generated images evert 2000 epochs, to see the changes in loss and generated images.

[11]:
```
def train(epochs, batch_size = 64, n_critic = 5):
  loss_dtrue = []
  loss_dfake = []
  loss_d = []
  loss_g = []


  y_train_real, y_train_fake = (-1)*np.ones((batch_size, 1)),  np.
↪ones((batch_size, 1))
  #y_train_real = np.array(y_train_real, dtype = 'float32')
  #y_train_fake = np.array(y_train_fake, dtype = 'float32')
  #y_eval_real,  y_eval_fake  = make_labels(eval_size)

  for e in range(epochs):
    d1_l = []
    d2_l = []

    for i in range(n_critic):
      ### select
      idx = np.random.randint(0, x_train.shape[0], batch_size)
      x_batch_real = x_train[idx] #true

      z = np.random.normal(loc = 0.0, scale = 1.0, size = (batch_size, 100))␣
↪#noise
      x_batch_fake = generator.predict(z)

      d1 = discriminator.train_on_batch(x_batch_real, y_train_real)
      d2 = discriminator.train_on_batch(x_batch_fake, y_train_fake)
      d1_l.append(d1)
      d2_l.append(d2)
```

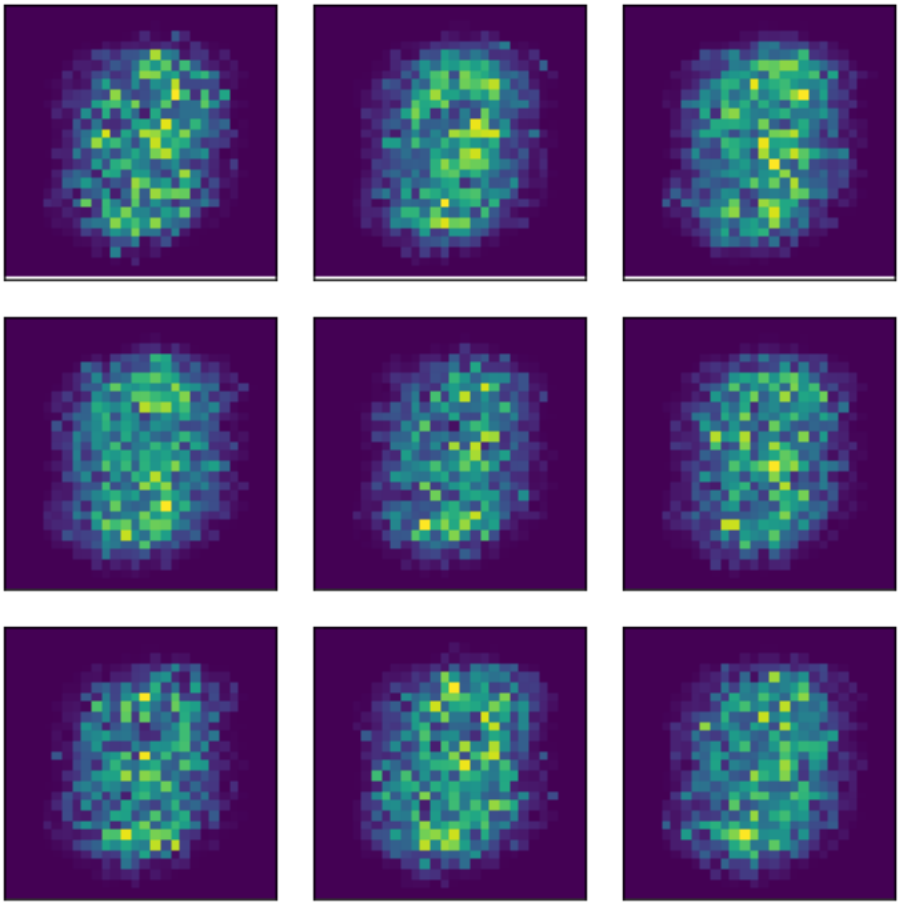
```
    for l in discriminator.layers:
                weights = l.get_weights()
                weights = [np.clip(w, -0.01, 0.01) for w in weights]
                l.set_weights(weights)



  d = 0.5 * np.mean(np.array(d2_l) + np.array(d1_l))

  #loss_dtrue.append(np.mean(d1_l))
  #loss_dfake.append(np.mean(d2_l))
  loss_d.append(d)
  #make_trainable(discriminator, False)
  #z2 = np.random.normal(loc = 0.0, scale = 1.0, size = (batch_size, 100))
  g_loss = wgan.train_on_batch(z, y_train_real)
  loss_g.append(g_loss)

  ##print progress
  if e % 500 == 0:
    print ("%d [D loss: %f] [G loss: %f]" % (e, d, g_loss))
  if e % 2000 == 0:
    np.random.seed(seed = 0)
    generated_digits = wgan.layers[0].predict(np.random.normal(0, 1, (9,␣
↪100)))
    ### Let's examine some sample images.
    sample_images = generated_digits

    plt.figure(figsize=(5, 5))
    for i in range(9):
      plt.subplot(3, 3, i+1)
      plt.imshow(sample_images[i].reshape(28, 28))
      plt.xticks([])
      plt.yticks([])
    plt.tight_layout()
    plt.show()


 #loss_d = loss_d1 + loss_d2
 return loss_d, loss_g, wgan
```

[12]:
```
result = train(50000)
```

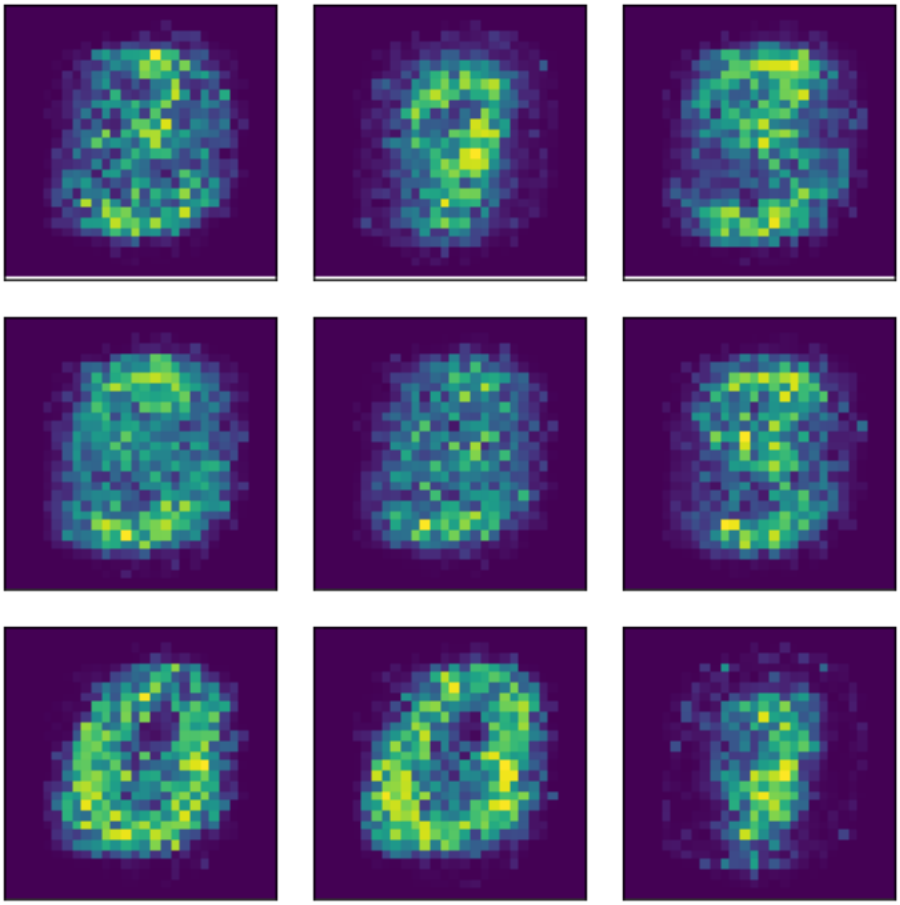
```
0 [D loss: 0.090387] [G loss: 0.000596]
```



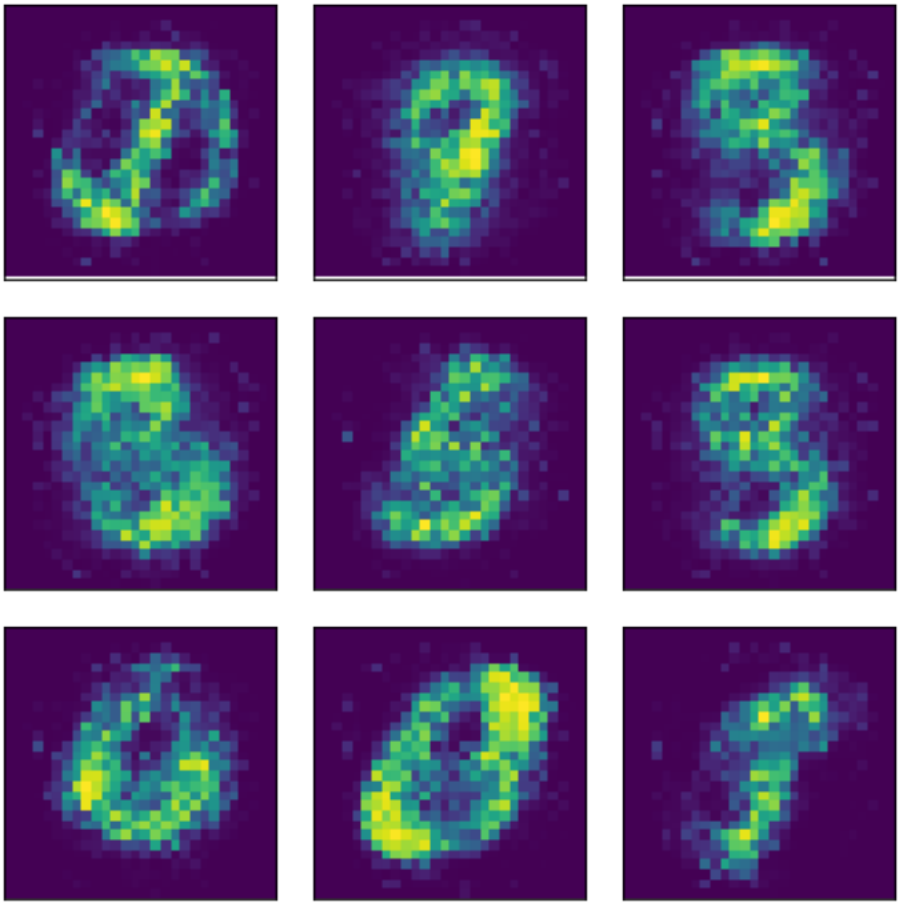
```
500 [D loss: -0.264239] [G loss: -2.736692]
1000 [D loss: 0.020696] [G loss: -2.686066]
1500 [D loss: 0.025777] [G loss: -2.026719]
2000 [D loss: 0.017423] [G loss: -1.412508]
```



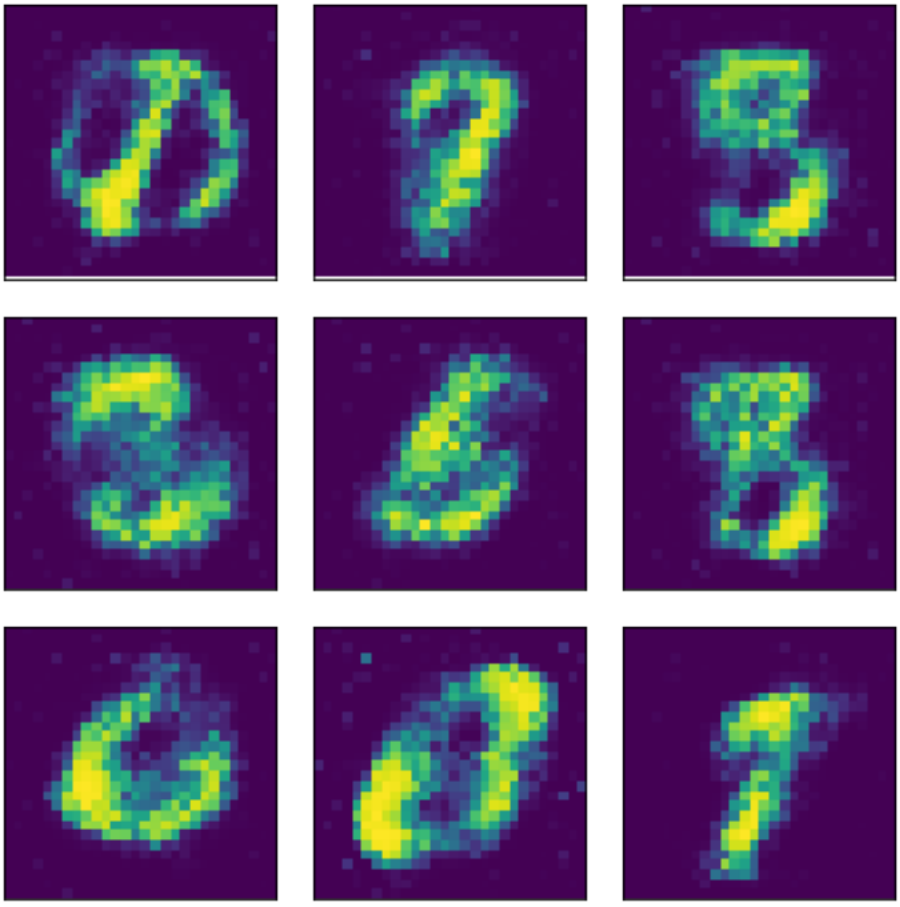
```
2500 [D loss: 0.012251] [G loss: -0.885673]
3000 [D loss: -0.018181] [G loss: -0.504861]
3500 [D loss: -0.035211] [G loss: -0.386890]
4000 [D loss: -0.051349] [G loss: -0.049106]
```



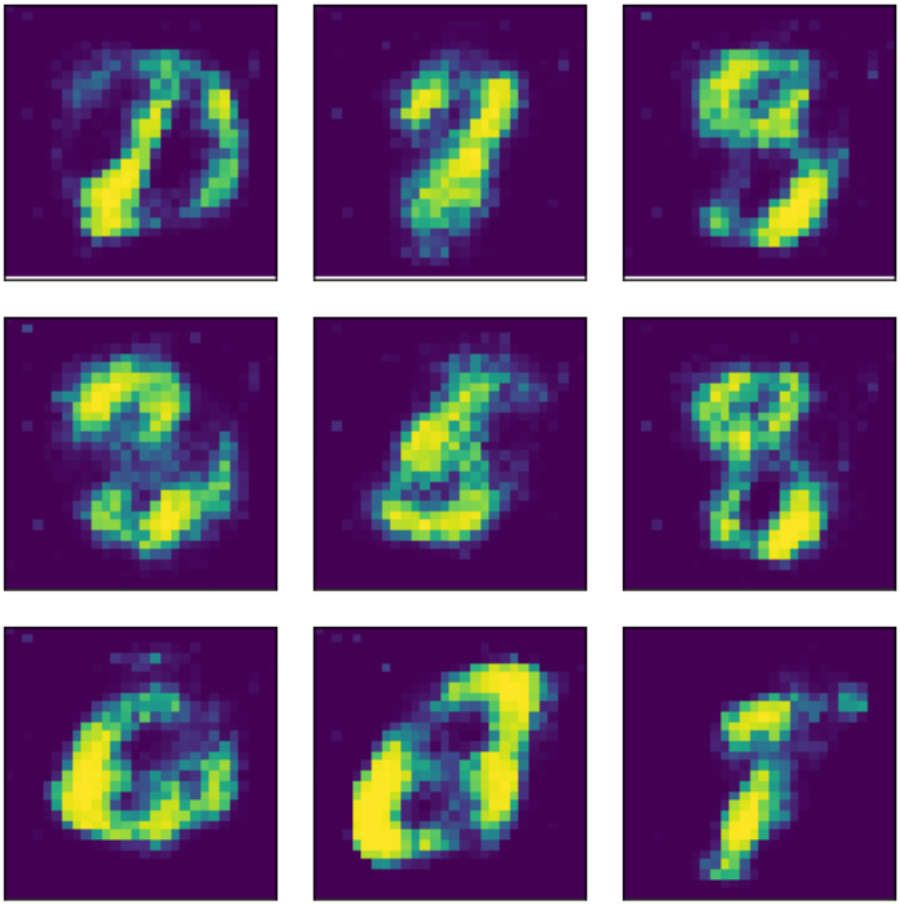
```
4500 [D loss: -0.072895] [G loss: -0.048554]
5000 [D loss: -0.055516] [G loss: -0.060816]
5500 [D loss: -0.053893] [G loss: -0.043105]
6000 [D loss: -0.040092] [G loss: -0.069191]
```



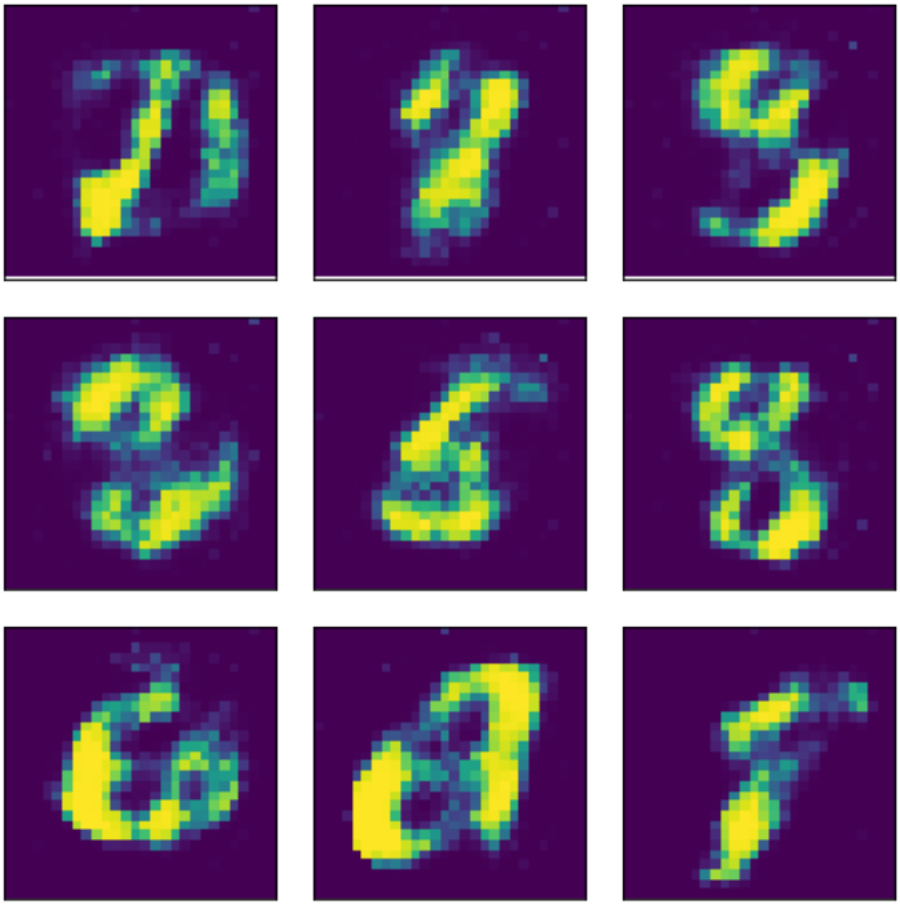
```
6500 [D loss: -0.040021] [G loss: -0.042686]
7000 [D loss: -0.036686] [G loss: -0.041328]
7500 [D loss: -0.030582] [G loss: -0.037648]
8000 [D loss: -0.029398] [G loss: -0.037225]
```



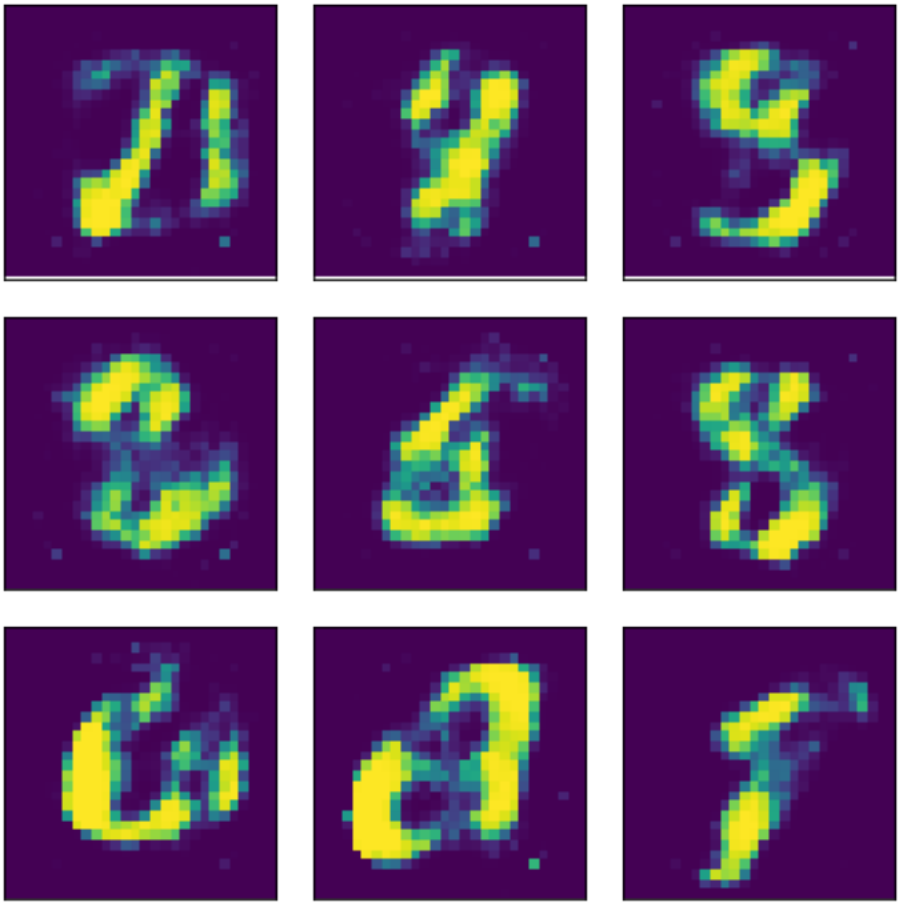
```
8500 [D loss: -0.024154] [G loss: -0.036759]
9000 [D loss: -0.025630] [G loss: -0.029136]
9500 [D loss: -0.025913] [G loss: -0.026307]
10000 [D loss: -0.018069] [G loss: -0.028535]
```



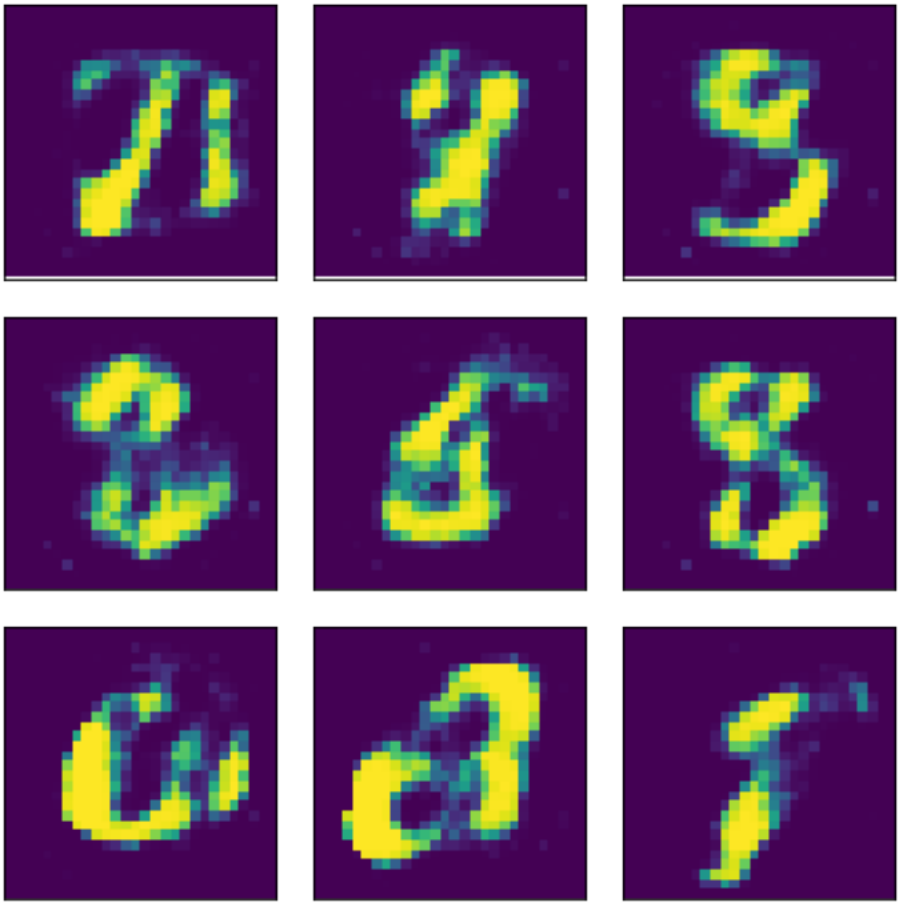
```
10500 [D loss: -0.018722] [G loss: -0.026016]
11000 [D loss: -0.018631] [G loss: -0.023853]
11500 [D loss: -0.016606] [G loss: -0.023687]
12000 [D loss: -0.015645] [G loss: -0.020861]
```



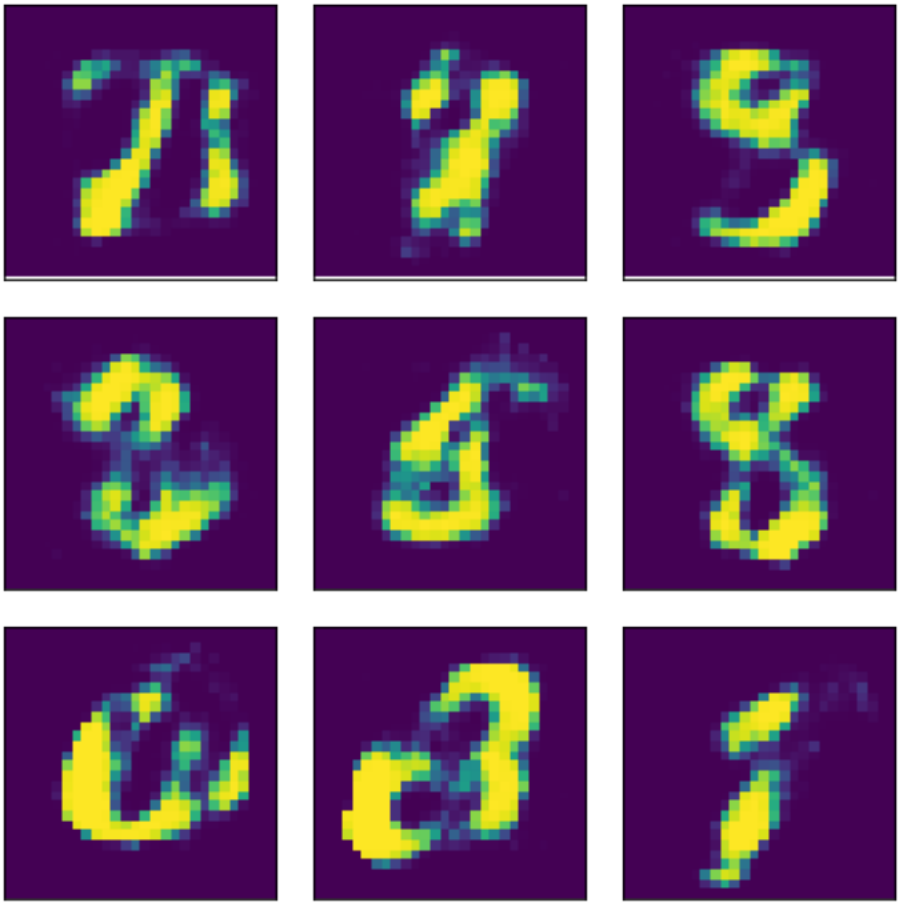
```
12500 [D loss: -0.015247] [G loss: -0.021685]
13000 [D loss: -0.014312] [G loss: -0.019753]
13500 [D loss: -0.012732] [G loss: -0.020535]
14000 [D loss: -0.011092] [G loss: -0.016984]
```



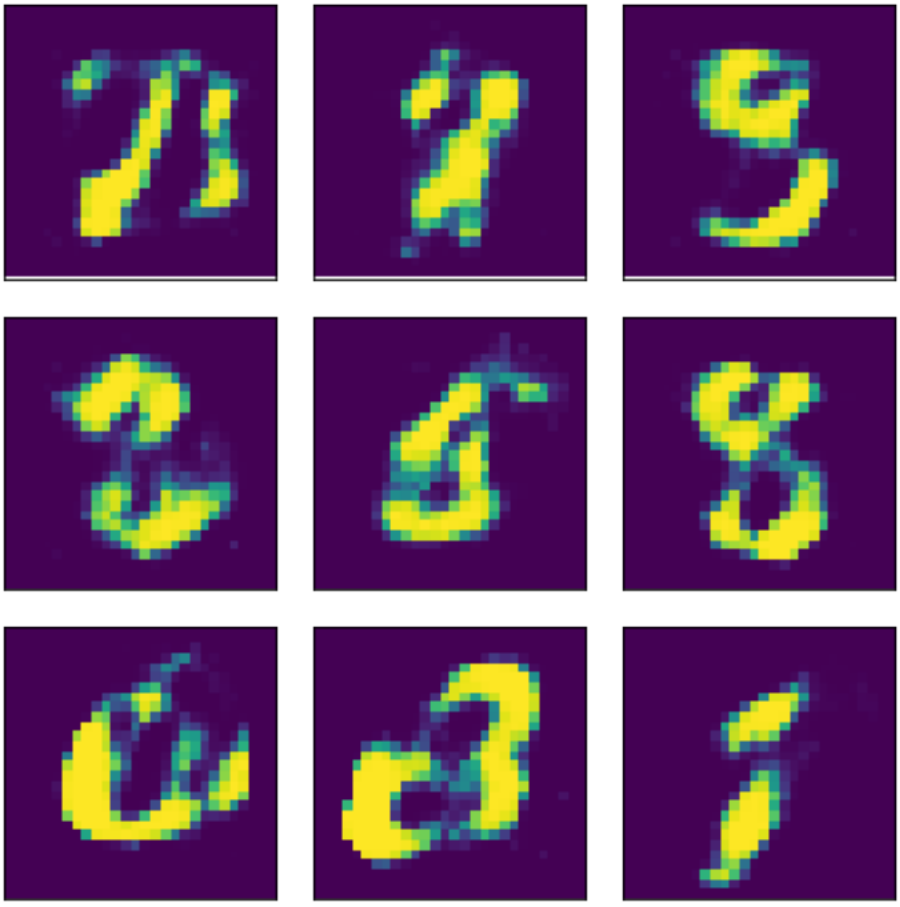
```
14500 [D loss: -0.011454] [G loss: -0.016784]
15000 [D loss: -0.011511] [G loss: -0.015056]
15500 [D loss: -0.011302] [G loss: -0.014420]
16000 [D loss: -0.009422] [G loss: -0.012803]
```



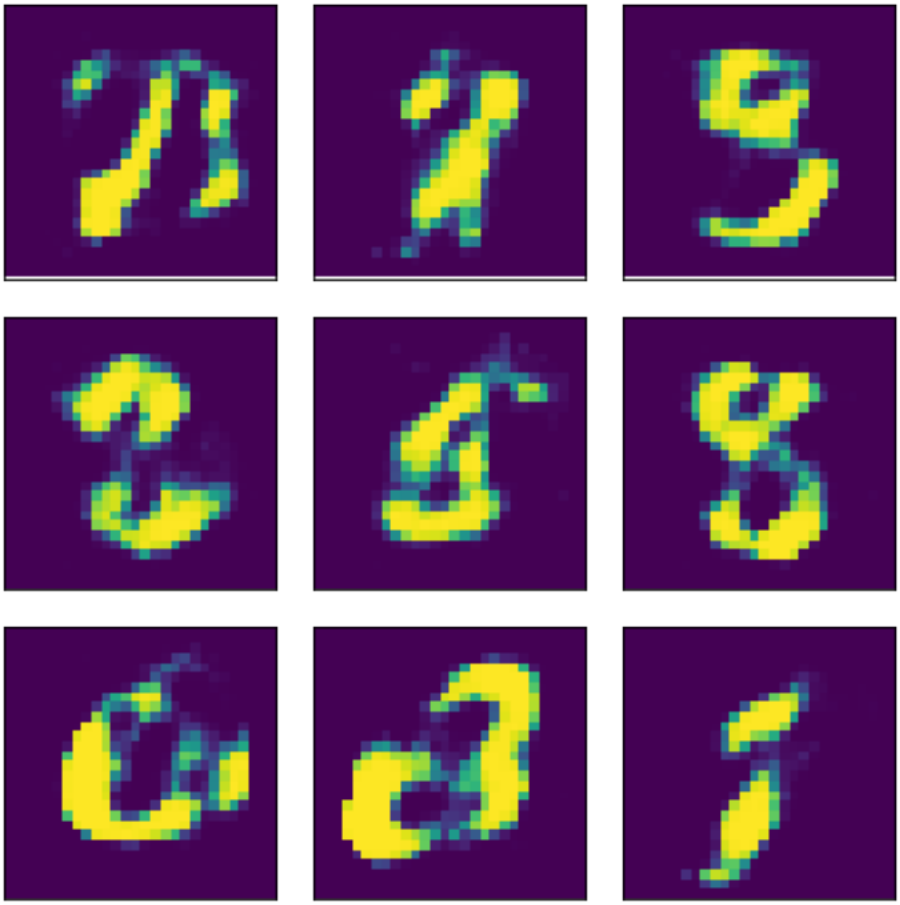
```
16500 [D loss: -0.009995] [G loss: -0.012544]
17000 [D loss: -0.009820] [G loss: -0.011872]
17500 [D loss: -0.010002] [G loss: -0.011404]
18000 [D loss: -0.008192] [G loss: -0.010556]
```



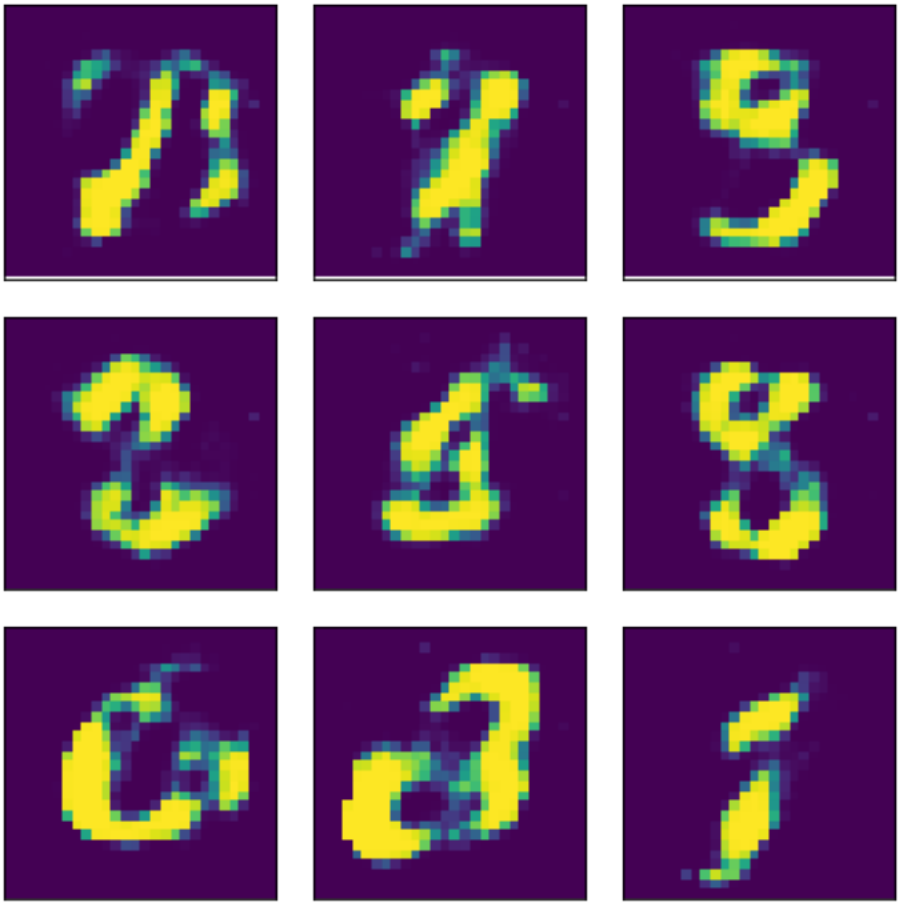
```
18500 [D loss: -0.008801] [G loss: -0.010349]
19000 [D loss: -0.008343] [G loss: -0.011062]
19500 [D loss: -0.008554] [G loss: -0.009084]
20000 [D loss: -0.007225] [G loss: -0.008581]
```



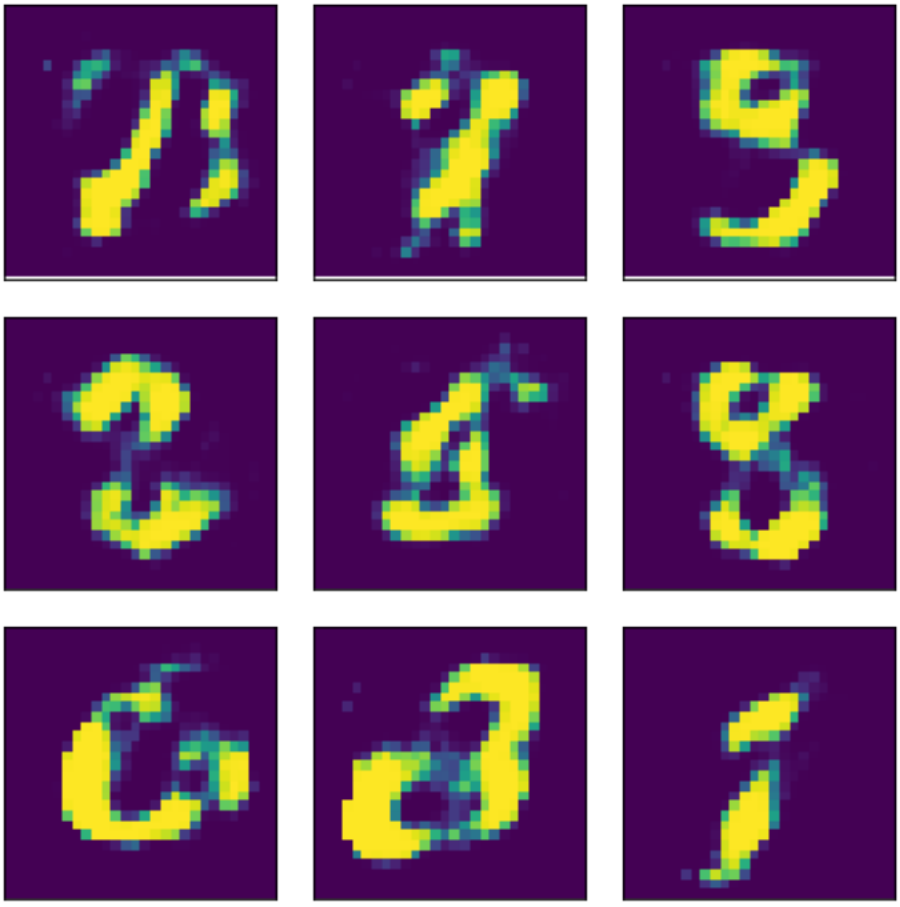
```
20500 [D loss: -0.008098] [G loss: -0.007488]
21000 [D loss: -0.007512] [G loss: -0.009000]
21500 [D loss: -0.007392] [G loss: -0.008707]
22000 [D loss: -0.006550] [G loss: -0.007076]
```



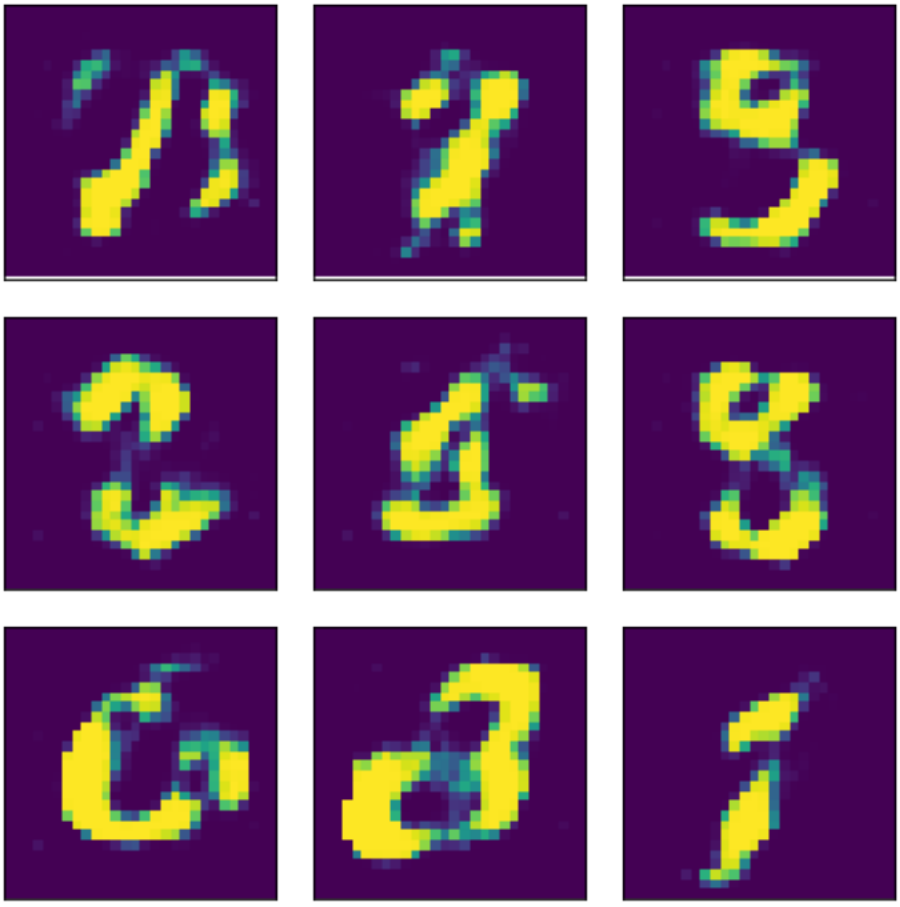
```
22500 [D loss: -0.007287] [G loss: -0.007684]
23000 [D loss: -0.006786] [G loss: -0.007946]
23500 [D loss: -0.006779] [G loss: -0.007871]
24000 [D loss: -0.006304] [G loss: -0.006290]
```



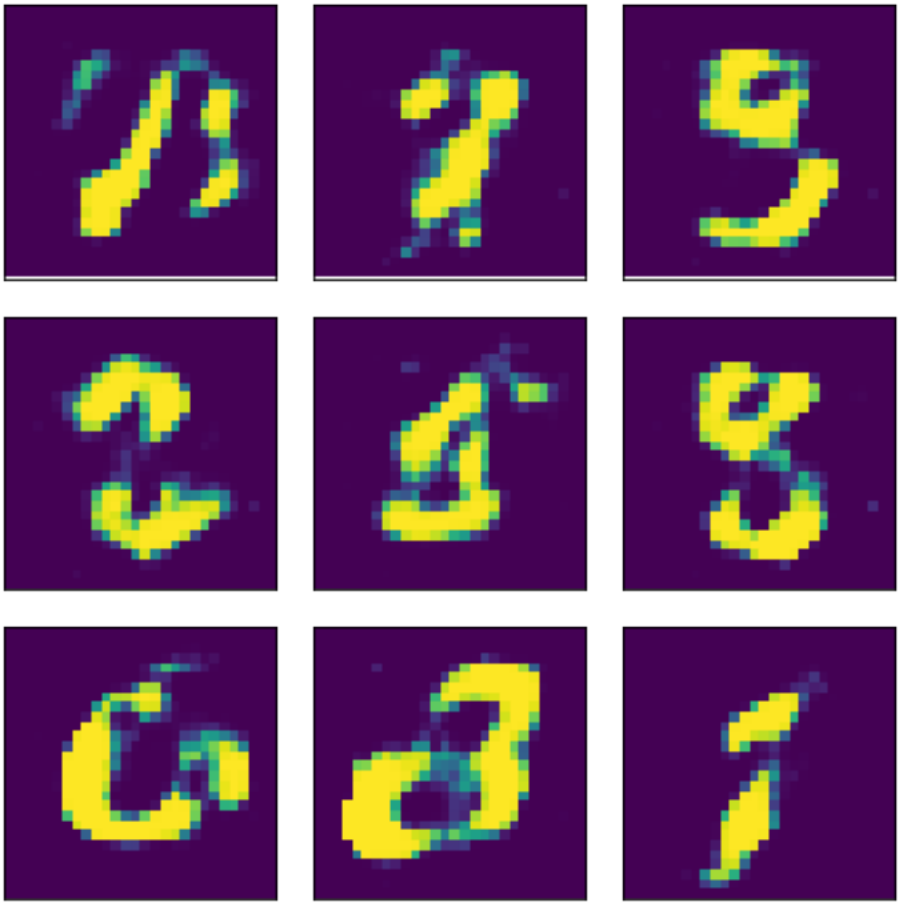
```
24500 [D loss: -0.006846] [G loss: -0.006571]
25000 [D loss: -0.006114] [G loss: -0.006652]
25500 [D loss: -0.006367] [G loss: -0.006541]
26000 [D loss: -0.006033] [G loss: -0.004798]
```



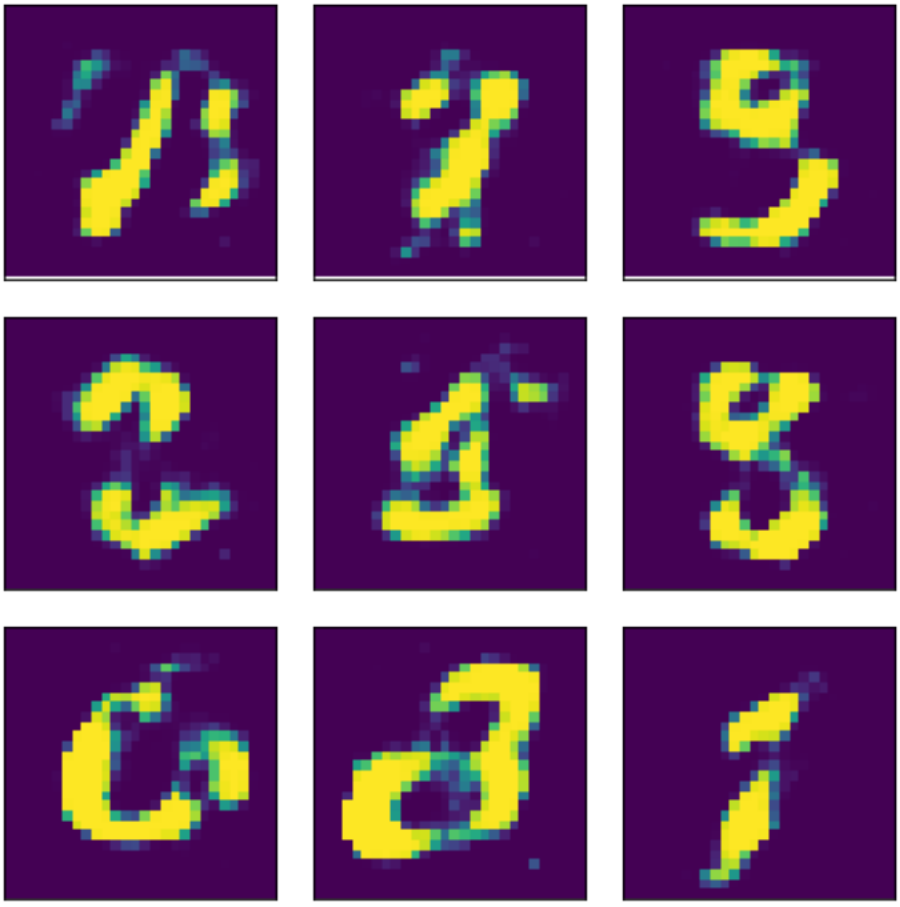
```
26500 [D loss: -0.006548] [G loss: -0.004722]
27000 [D loss: -0.005715] [G loss: -0.005965]
27500 [D loss: -0.005870] [G loss: -0.005767]
28000 [D loss: -0.005658] [G loss: -0.003786]
```



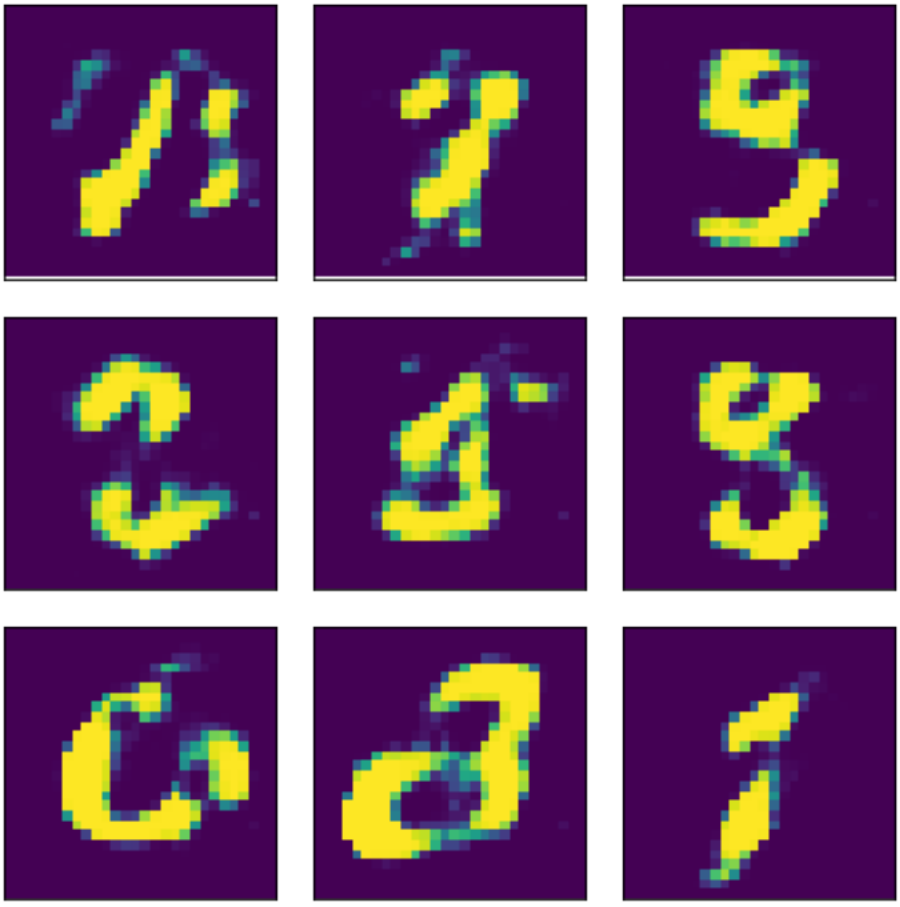
```
28500 [D loss: -0.006283] [G loss: -0.004185]
29000 [D loss: -0.005306] [G loss: -0.005197]
29500 [D loss: -0.005398] [G loss: -0.004972]
30000 [D loss: -0.005485] [G loss: -0.003208]
```



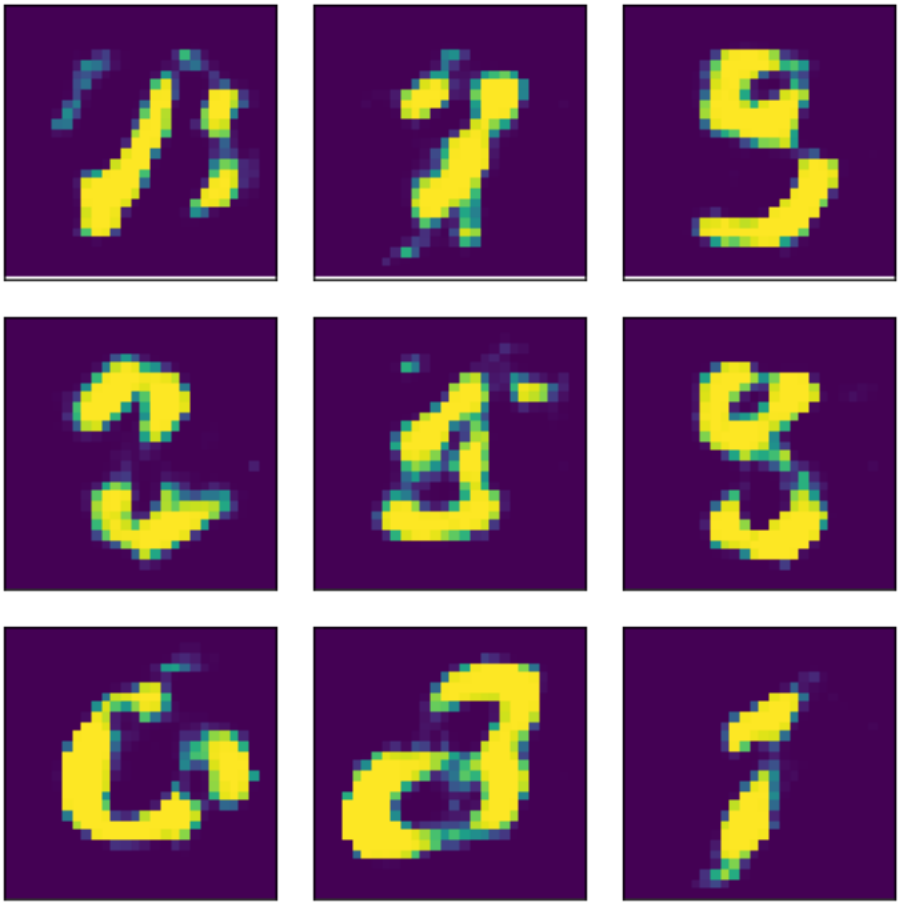
```
30500 [D loss: -0.006023] [G loss: -0.003293]
31000 [D loss: -0.005087] [G loss: -0.003934]
31500 [D loss: -0.005104] [G loss: -0.004487]
32000 [D loss: -0.005238] [G loss: -0.002615]
```



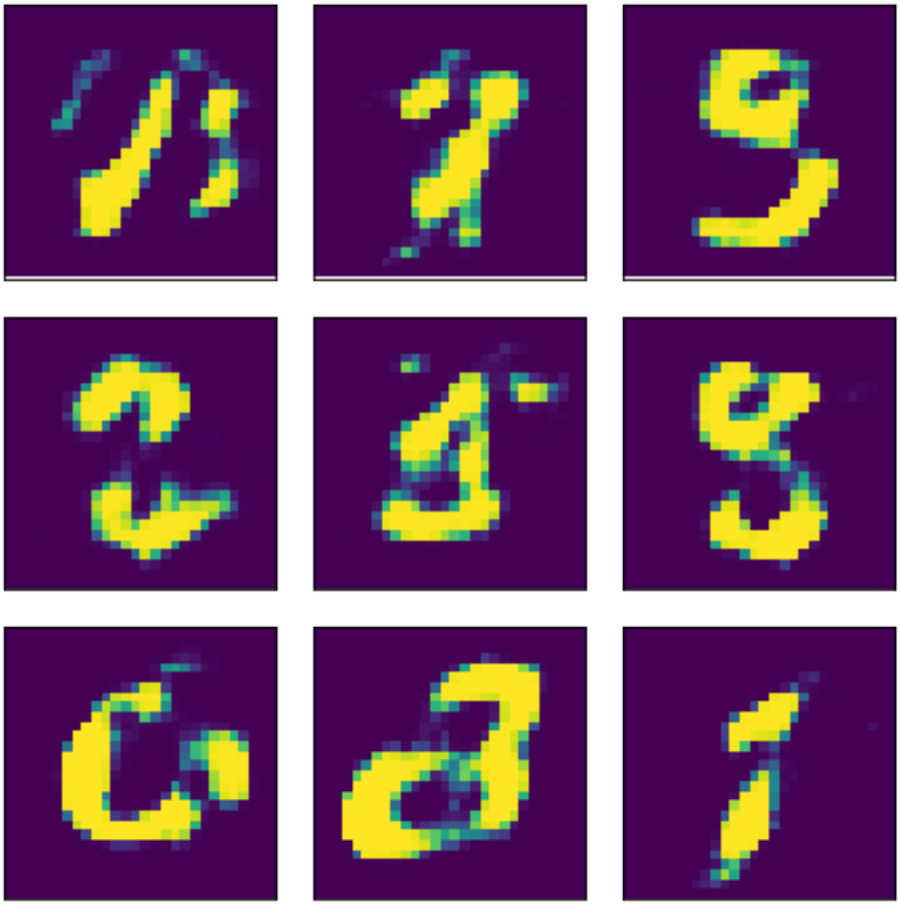
```
32500 [D loss: -0.005761] [G loss: -0.002809]
33000 [D loss: -0.004923] [G loss: -0.003040]
33500 [D loss: -0.005021] [G loss: -0.004182]
34000 [D loss: -0.005107] [G loss: -0.002298]
```



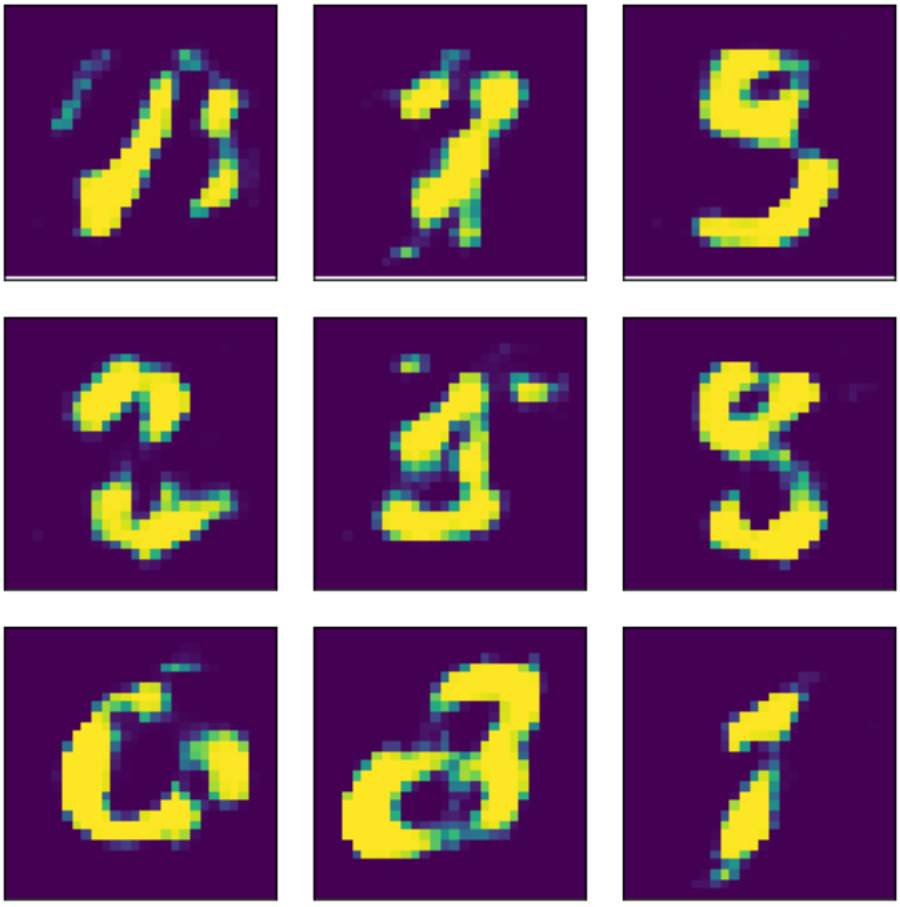
```
34500 [D loss: -0.005612] [G loss: -0.002983]
35000 [D loss: -0.004708] [G loss: -0.003030]
35500 [D loss: -0.004852] [G loss: -0.004192]
36000 [D loss: -0.004869] [G loss: -0.001946]
```



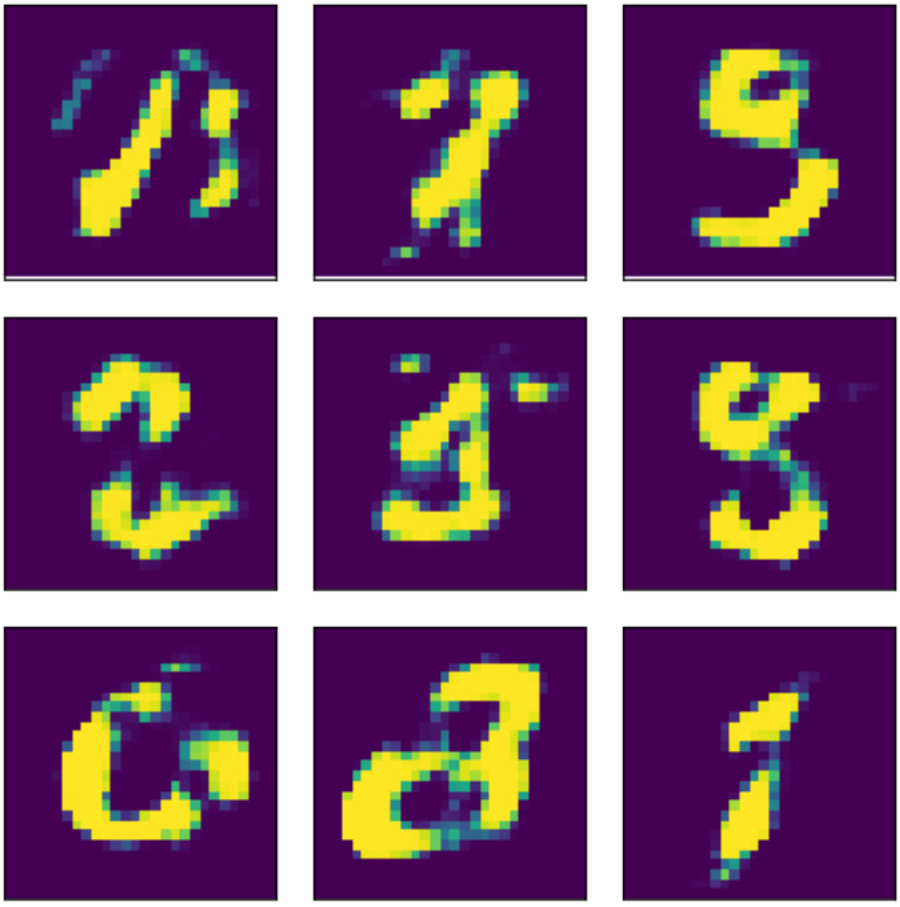
```
36500 [D loss: -0.005422] [G loss: -0.002186]
37000 [D loss: -0.004479] [G loss: -0.002379]
37500 [D loss: -0.004588] [G loss: -0.003696]
38000 [D loss: -0.004727] [G loss: -0.001565]
```



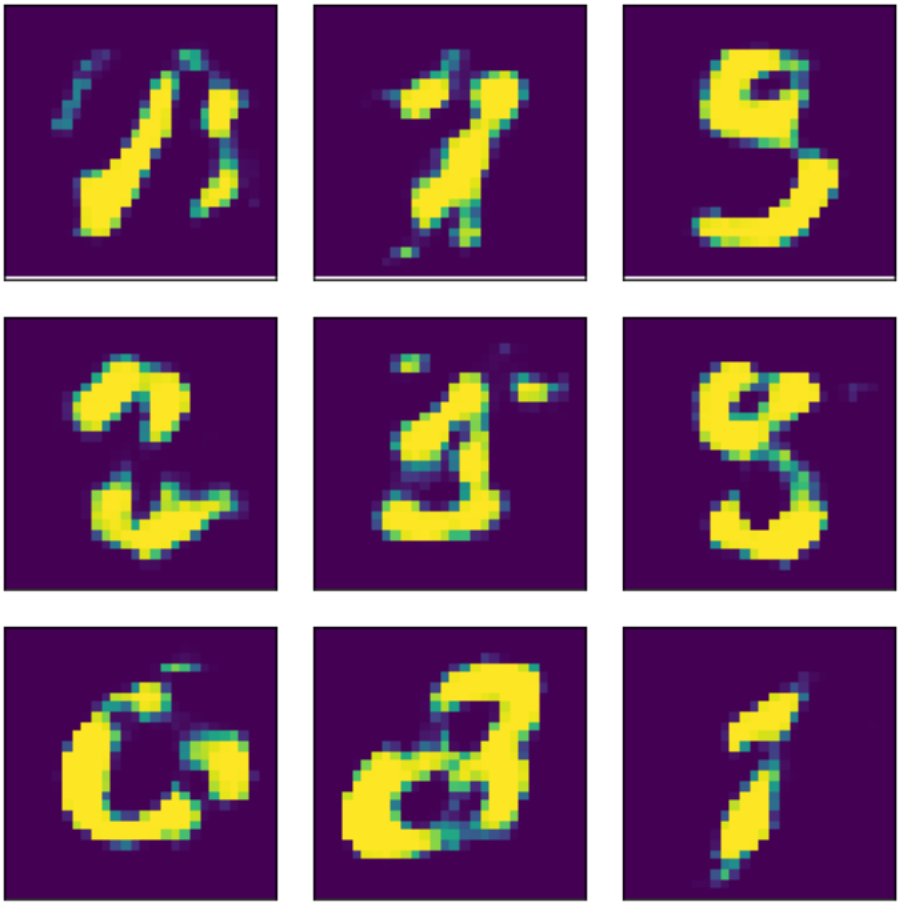
```
38500 [D loss: -0.005184] [G loss: -0.001689]
39000 [D loss: -0.004250] [G loss: -0.001895]
39500 [D loss: -0.004434] [G loss: -0.002523]
40000 [D loss: -0.004515] [G loss: -0.000940]
```



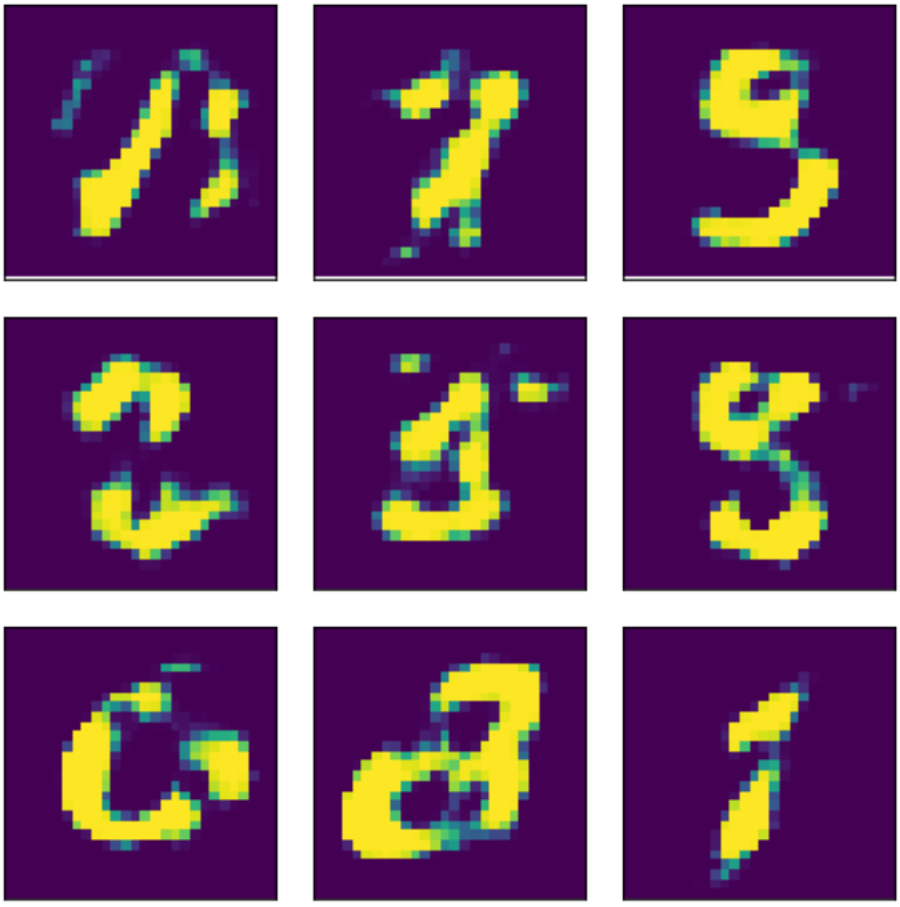
```
40500 [D loss: -0.004976] [G loss: -0.001378]
41000 [D loss: -0.004063] [G loss: -0.001686]
41500 [D loss: -0.004236] [G loss: -0.002265]
42000 [D loss: -0.004368] [G loss: -0.000515]
```



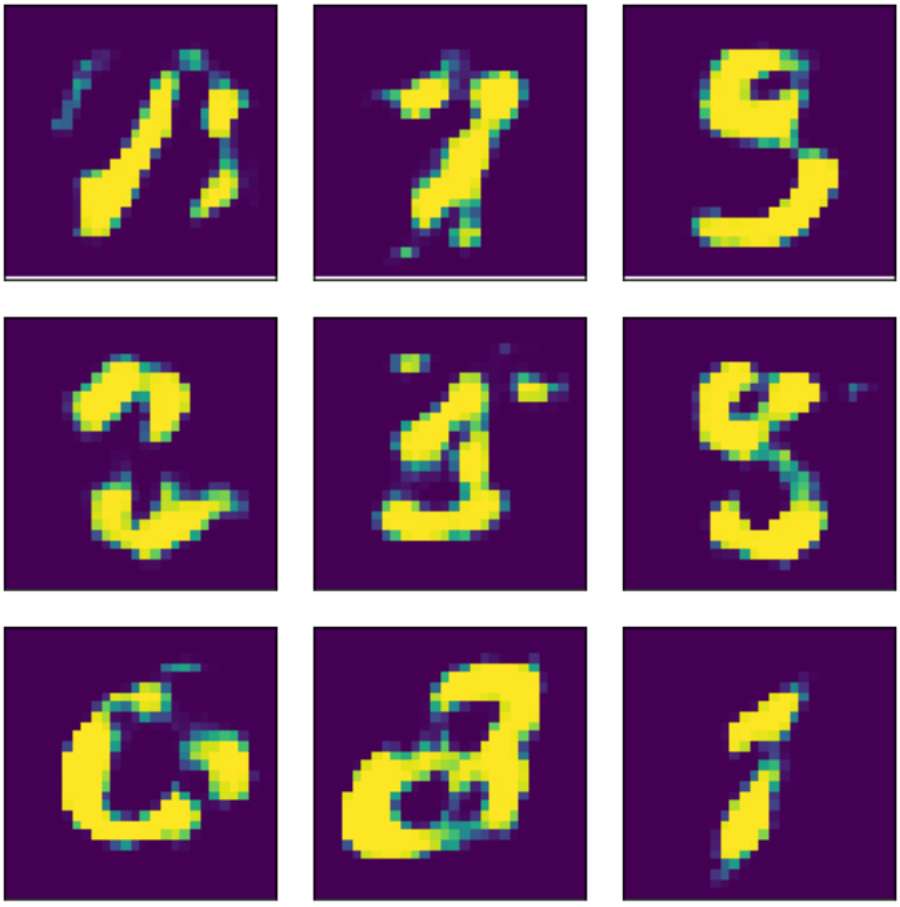
```
42500 [D loss: -0.004795] [G loss: -0.000937]
43000 [D loss: -0.003953] [G loss: -0.001536]
43500 [D loss: -0.004218] [G loss: -0.001741]
44000 [D loss: -0.004171] [G loss: -0.000338]
```



```
44500 [D loss: -0.004608] [G loss: -0.001175]
45000 [D loss: -0.003812] [G loss: -0.001211]
45500 [D loss: -0.004009] [G loss: -0.001619]
46000 [D loss: -0.003987] [G loss: -0.000151]
```

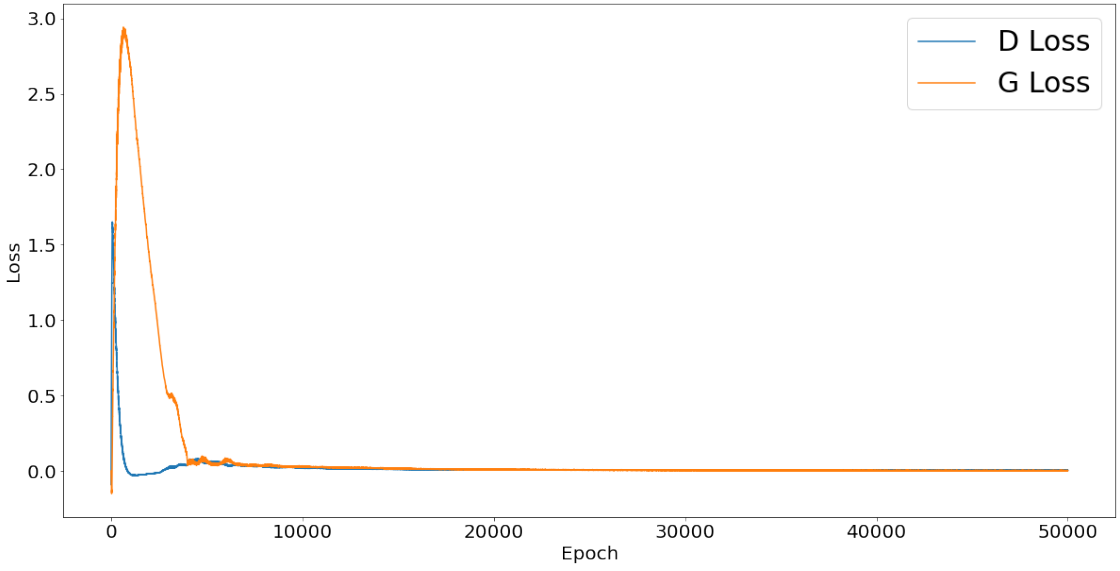


```
46500 [D loss: -0.004526] [G loss: -0.000913]
47000 [D loss: -0.003801] [G loss: -0.001294]
47500 [D loss: -0.003861] [G loss: -0.001758]
48000 [D loss: -0.003764] [G loss: -0.000376]
```



```
48500 [D loss: -0.004415] [G loss: -0.001061]
49000 [D loss: -0.003795] [G loss: -0.000907]
49500 [D loss: -0.003830] [G loss: -0.001748]
```

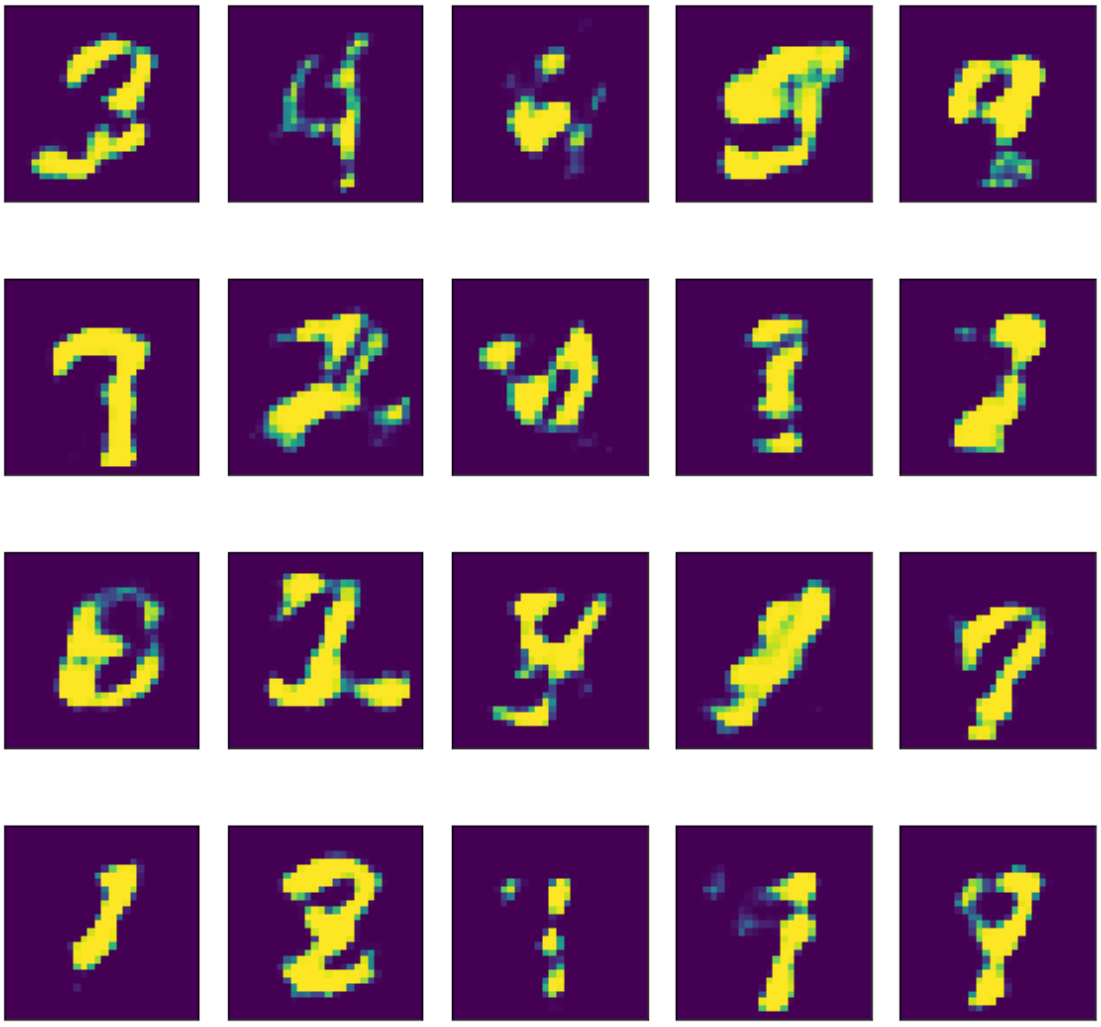
[16]:
```
##plot the loss
plt.figure(figsize = (20, 10))
plt.plot(range(1, 50001), (-1) * np.array(result[0]), label = 'D Loss')
plt.plot(range(1, 50001), (-1) * np.array(result[1]), label = 'G Loss')
plt.xlabel('Epoch', fontsize = 20)
plt.ylabel('Loss', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

plt.legend(fontsize = 30)


plt.show()
```



0.8 Result

Here, I show the generated images after training 50000 epochs.

[14]:
```
generated_digits = result[2].layers[0].predict(np.random.normal(0, 1, (20,␣
↪100)))

### Let's examine some sample images.
np.random.seed(seed = 0)
sample_images = generated_digits

plt.figure(figsize=(8, 8))
for i in range(20):
  plt.subplot(4, 5, i+1)
  plt.imshow(sample_images[i].reshape(28, 28))
  plt.xticks([])
  plt.yticks([])
plt.tight_layout()
plt.show()
```



0.9 Discussion

Based on the images I generated, I think more research needs to be done in the future, such as changing the architecture of networks, tuning the hyperparameter such as learning rate, etc.

wgan_svhn

December 21, 2020

0.1 5 Wasserstein GAN on SVHN

We use the same structure as DCGAN on SVHN, except from the following part:
* Use RMSprop as optimizer * Use Wesserstein loss * Delete sigmoid in the last layer of discriminator * Set weight clipping

[22]:
```
from scipy.io import loadmat
import tensorflow as tf
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras.layers import Dense
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers
from tensorflow.keras.layers import Input, Conv2D, Dense, BatchNormalization,␣
↪LeakyReLU, Reshape, Conv2DTranspose, Dropout, Flatten, AveragePooling2D, ReLU
from tensorflow.keras.layers import Dropout
from tensorflow.keras.constraints import Constraint
import PIL
import time
from IPython import display
import datetime
import pickle as pkl

from scipy.io import loadmat

BATCH_SIZE = 64
LEARNING_RATE = 5e-5
ALPHA_WEIGHT = 200
DISCRIMINATOR_FACTOR = 2
```

[2]:
```
%load_ext tensorboard
!rm -rf ./logs/wgan_svhn_gradient_tape/
```

[3]:
```
!rm -rf ./svhn_data/
!mkdir svhn_data
```

```
from urllib.request import urlretrieve
from os.path import isfile, isdir
from tqdm import tqdm

data_dir = 'svhn_data/'

if not isdir(data_dir):
  raise Exception("Data directory doesn't exist!")

class DLProgress(tqdm):
  last_block = 0

  def hook(self, block_num=1, block_size=1, total_size=None):
        self.total = total_size
        self.update((block_num - self.last_block) * block_size)
        self.last_block = block_num

if not isfile(data_dir + "train_32x32.mat"):
  with DLProgress(unit='B', unit_scale=True, miniters=1, desc='SVHN Training␣
 ↪Set') as pbar:
    urlretrieve(
            'http://ufldl.stanford.edu/housenumbers/train_32x32.mat',
            data_dir + 'train_32x32.mat',
            pbar.hook)

train_images = loadmat(data_dir + 'train_32x32.mat')['X']
```

```
SVHN Training Set: 182MB [00:11, 16.1MB/s]
```

[4]:
```
train_images = np.rollaxis(train_images, 3).astype('float32')
train_images = (train_images - 127.5) / 127.5 # Normalize the images to [-1, 1]
```

[5]:
```
train_images.shape
```

[5]:
```
(73257, 32, 32, 3)
```

[6]:
```
train_dataset = tf.data.Dataset.from_tensor_slices(train_images)
train_dataset = train_dataset.shuffle(60000).batch(BATCH_SIZE)
```

Unlike gan, we use RMSprop(learning_rate = 0.00005) as optimizer, and use wesserstein loss instead of binary_crossentropy.

[7]:
```
def make_generator_model():
  model = tf.keras.Sequential()
  model.add(layers.Dense(4*4*512, use_bias=False, input_shape=(100,)))
  model.add(layers.BatchNormalization())
  model.add(layers.LeakyReLU(alpha=0))
```

```
    # model.add(layers.Dropout(0.2))
  model.add(layers.Reshape((4, 4, 512)))
  assert model.output_shape == (None, 4, 4, 512) # Note: None is the batch size
  model.add(layers.Conv2DTranspose(256, (5, 5), strides=(2, 2), padding='same',␣
↪use_bias=False))
  assert model.output_shape == (None, 8, 8, 256)
  model.add(layers.BatchNormalization())
  model.add(layers.LeakyReLU(alpha=0))
  model.add(layers.Conv2DTranspose(128, (5, 5), strides=(2, 2), padding='same',␣
↪use_bias=False))
  assert model.output_shape == (None, 16, 16, 128)
  model.add(layers.BatchNormalization())
  model.add(layers.LeakyReLU(alpha=0))
  model.add(layers.Conv2DTranspose(3, (5, 5), strides=(2, 2), padding='same',␣
↪use_bias=False, activation='tanh'))
  assert model.output_shape == (None, 32, 32, 3)
  return model
generator = make_generator_model()
generator_optimizer = tf.keras.optimizers.RMSprop(learning_rate = 0.00005)
```

[8]:
```
noise = tf.random.uniform([1, 100], -1, 1)
generated_image = generator(noise, training=False)
```

In the last layer of discriminator, we delete sigmoid.

[9]:
```
def make_discriminator_model():
  model = tf.keras.Sequential()
  model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
         input_shape=[32, 32, 3]))
  model.add(layers.LeakyReLU(alpha=0.2))

  model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
  model.add(layers.BatchNormalization())
  model.add(layers.LeakyReLU(alpha=0.2))

  model.add(layers.Conv2D(256, (5, 5), strides=(2, 2), padding='same'))
  model.add(layers.BatchNormalization())
  model.add(layers.LeakyReLU(alpha=0.2))

  model.add(layers.Flatten())
  model.add(layers.Dense(1))

  return model
discriminator = make_discriminator_model()
discriminator_optimizer = tf.keras.optimizers.RMSprop(learning_rate = 0.00005)
```

Unlike DCGAN, we use Wesserstein loss.

[10]:
```
def discriminator_loss(real_output, fake_output):
  real_loss = tf.reduce_mean(real_output)
  fake_loss = tf.reduce_mean(fake_output)
  total_loss = fake_loss - real_loss
  return total_loss
```

[11]:
```
def generator_loss(fake_output):
  x = -tf.reduce_mean(fake_output)
  return x
```

[12]:
```
EPOCHS = 30
noise_dim = 100
num_examples_to_generate = 80
seed = tf.random.uniform([num_examples_to_generate, noise_dim], -1, 1)
```

[13]:
```
@tf.function
def train_generator():
  noise = tf.random.uniform([BATCH_SIZE, noise_dim], minval=-1, maxval=1)
  with tf.GradientTape() as gen_tape:
    generated_images = generator(noise, training=True)
    fake_output = discriminator(generated_images, training=True)
    gen_loss = generator_loss(fake_output)

  gradients_of_generator = gen_tape.gradient(gen_loss, generator.
 ↪trainable_variables)
  generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.
 ↪trainable_variables))
  return gen_loss
```

[17]:
```
@tf.function
def train_discriminator(images):
  noise = tf.random.uniform([BATCH_SIZE, noise_dim], minval=-1, maxval=1)
  with tf.GradientTape() as disc_tape:
    generated_images = generator(noise, training=True)

    real_output = discriminator(images, training=True)
    fake_output = discriminator(generated_images, training=True)

    disc_loss = discriminator_loss(real_output, fake_output)

  gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.
 ↪trainable_variables)
  discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,␣
 ↪discriminator.trainable_variables))

  _ = [p.assign(tf.clip_by_value(p, -0.01, 0.01)) for p in discriminator.
 ↪trainable_variables]
```

```
  return disc_loss
```

Set weight clipping From -0.01 to 0.01.

[19]:
```
def generate_and_save_images(model, epoch, test_input):
  predictions = model(test_input, training=False)

  fig = plt.figure(figsize=(10,8))

  for i in range(predictions.shape[0]):
    plt.subplot(predictions.shape[0]//10, 10, i+1)
    plt.imshow(predictions[i, :, :, :] * .5 + .5)
    plt.axis('off')

  plt.show()
def show_losses(losses):
  losses = np.array(losses)

  fig, ax = plt.subplots()
  plt.plot(losses.T[0], label='Discriminator')
  plt.plot(losses.T[1], label='Generator')
  plt.title("Validation Losses(Wesserstein loss)")
  plt.legend()
  plt.show()
```

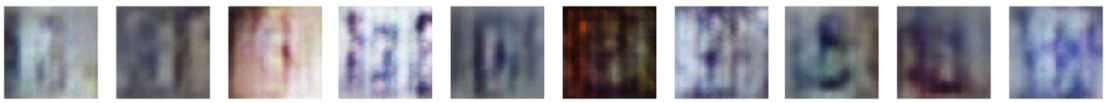
[24]:
```
def train(dataset, epochs):
  losses = []
  for epoch in range(epochs):
    for image_batch in dataset:
      for _ in range(5):
        disc_loss = train_discriminator(image_batch)
      gen_loss = train_generator()

    losses.append((disc_loss, gen_loss))
    print("Epoch: {:>3}/{} Discriminator Loss: {:>7.4f}  Generator Loss: {:>7.
 ↪4f}".format(epoch+1, epochs, disc_loss, gen_loss))
    if (epoch + 1) % 5 == 0:
      seed = tf.random.uniform([num_examples_to_generate, noise_dim], -1, 1)
      generate_and_save_images(generator,epoch + 1,seed[:10])

  show_losses(losses)
  generate_and_save_images(generator,epochs,seed)
```

[25]:
```
train(train_dataset, EPOCHS)
```

```
Epoch:   1/30 Discriminator Loss: -0.0128  Generator Loss:  0.0550
Epoch:   2/30 Discriminator Loss: -0.0162  Generator Loss:  0.0475
```

```
Epoch:   3/30 Discriminator Loss: -0.0383  Generator Loss:  0.0116
Epoch:   4/30 Discriminator Loss: -0.0160  Generator Loss: -0.0121
Epoch:   5/30 Discriminator Loss: -0.0566  Generator Loss:  0.0520
```

```
Epoch:   6/30 Discriminator Loss: -0.0104  Generator Loss: -0.0177
Epoch:   7/30 Discriminator Loss: -0.0533  Generator Loss:  0.0230
Epoch:   8/30 Discriminator Loss: -0.0296  Generator Loss:  0.0682
Epoch:   9/30 Discriminator Loss: -0.0389  Generator Loss:  0.0439
Epoch:  10/30 Discriminator Loss: -0.1105  Generator Loss: -0.0175
```



```
Epoch:  11/30 Discriminator Loss: -0.0363  Generator Loss: -0.0232
Epoch:  12/30 Discriminator Loss: -0.0449  Generator Loss: -0.0570
Epoch:  13/30 Discriminator Loss: -0.0952  Generator Loss:  0.0363
Epoch:  14/30 Discriminator Loss: -0.0422  Generator Loss:  0.0937
Epoch:  15/30 Discriminator Loss:  0.0132  Generator Loss:  0.0022
```

```
Epoch:  16/30 Discriminator Loss: -0.0715  Generator Loss: -0.0164
Epoch:  17/30 Discriminator Loss: -0.0124  Generator Loss:  0.0236
Epoch:  18/30 Discriminator Loss: -0.0738  Generator Loss:  0.0266
Epoch:  19/30 Discriminator Loss: -0.0683  Generator Loss:  0.0807
Epoch:  20/30 Discriminator Loss: -0.0525  Generator Loss: -0.0270
```

```
Epoch:  21/30 Discriminator Loss:  0.0395  Generator Loss: -0.0579
Epoch:  22/30 Discriminator Loss: -0.0652  Generator Loss: -0.0397
Epoch:  23/30 Discriminator Loss: -0.0415  Generator Loss: -0.0077
Epoch:  24/30 Discriminator Loss: -0.1022  Generator Loss: -0.0009
Epoch:  25/30 Discriminator Loss: -0.1251  Generator Loss:  0.0001
```

```
Epoch:  26/30 Discriminator Loss: -0.1029  Generator Loss:  0.0116
Epoch:  27/30 Discriminator Loss: -0.0015  Generator Loss:  0.0226
Epoch:  28/30 Discriminator Loss: -0.1414  Generator Loss: -0.0604
Epoch:  29/30 Discriminator Loss:  0.0137  Generator Loss: -0.0045
Epoch:  30/30 Discriminator Loss: -0.1179  Generator Loss: -0.0385
```